今から始める Cocoa プログラミング
文書ファイルを扱うアプリケーションを作る

目次

(1)久しぶりにお届けします 2
(2)Application Kit の機能 5
(3)作成するアプリケーションの仕様 9
(4)プロジェクトの用意 16
(5)初期状態での動きを見る 26
(6)既存の nib ファイルの設定を見てみる 33
(7)文書ウインドウに NSTextView を配置する 38
(8)新規にウインドウを開くときの処理 46
NSData について 50
(9)NSTextView の使い方 52
(10)保存結果と文書ファイル 57
(11)ファイルを開く 61
(12)開いたときにカーソルを点滅 65
(13)Revert を組み込む 67
(14)Cocoa でのファイル情報 72
Cocoa でのファイル情報関連機能 73
NSDictionary クラス 76
(15)ファイルタイプとクリエイタを設定する 79
(16)Preferences の組み込み 83
環境設定のユーザインタフェース 83
環境設定を扱うクラスの設計 86
クラスを稼働させる 96
環境設定の永続記憶 101
環境設定値を記録する 105

(1)久しぶりにお届けします

久しぶりで「今から始める Cocoa プログラミング」の記事をお届けしたい。このシリーズを初めた頃の記事は、Mac OS X は Public Beta ベースだったために Project Builder のバージョンも古く、厳密には記事の通りに物事は進まなくなってしまっているのであるが、基本は変わらないということで、以前の記事も参考にしていただきたい。

Interface Builder を使ってユーザーインタフェースを作るというのは、いわばアプリケーション見ての通り状態なので、それこそなんとか使っているうちに分かる範囲だと思われる。むしろ、難しいのは、Interface Builder でのクラスの設計やオブジェクトのインスタンス化の部分である。そして、あるインスタンスで別のオブジェクトを参照するという Outlet や、あるいは何らかの動作で別のインスタンスのメソッドを呼び出すという action といった概念を理解することである。これまでの部分ではそうしたところを極力重点的に解説を行ってきたつもりである。

そして、もう 1 つのハードルは Cocoa というフレームワークの理解だろう。これは Cocoa には限らないが、大量にクラスが定義されていて大量にメソッドなどがあるわけで、いきなりクラスの仕様書を見て面喰らうのは誰でも同じである。ただ、プログラミングを行うにはそうしたところをどうしてもハードルとしてこえないといけないのであるが、いちばんポイントになる部分から攻めて行くことで、それなりに見通しが良くなれば、最初はさっぱり訳の分からなかったクラスのリファレンスも徐々に見えてくるというものである。

前回までは、単純な 1 ウインドウのアプリケーションを作ってみた。ある意味では単純なのであるが、ファイルオープンの機能を組み込むとなるとちょっと難しくなってくるという側面もあった。それで半年も間が空いてしまって大変申し訳ないのだが、改めてここで少し長めに推移すると思うが、文書ファイルを扱うアプリケーションを Cocoa のフレームワークで作る方法を解説したい。たとえば、エディタとかビューアとかそうしたものを作成する機能だ。なお、Cocoa については、MDOnline では鶴薗さんが「Cocoa はやっぱり！出張版」でも解説をお願いしていることもあるのだが、新居の連載の方では Java で開発することを基本に今後も続けさせていただくことにする。

そういうわけで、今回からしばらく「文書ファイルを扱うアプリケーションを作る」というテーマで、何度かに渡って記事をお届けするが、アプリケーションに対して要

請される機能についても、順次追加して、かなり引っ張って行くつもりである。だから、五月雨的に何ヶ月か続くということもあり得る点はご了承いただきたい。ベースになるのは、2001 年 12 月 9 日に、MOSA Software Meeting の「Cocoa-Java で何かアプリケーションを作ってみる」という講演である。あまりに中身のないタイトルに笑いとひんしゅくを買ってしまったかもしれないが、いろいろ事情があってタイトル決定までに時間がなかったというわけでお許しいただきたい。明確に「文書ファイルを扱うアプリケーションを作る」とタイトルを付ければよかったのだが、準備ができなかった場合も想定してちょっと逃げ道を造ってしまったと言う次第だ。申し訳ない。
セミナーに参加した方は分かるように、最終的に単にテキストエディタ作るだけのことであれば、1 時間もかからないで終わってしまう作業である。具体的には Cocoa の Application Kit にある NSDocument というクラスを中心にしたいくつかのクラスを利用することで、文書ファイルを利用するアプリケーションは作成できてしまう。ユーザインタフェースも Cocoa にあるものをそのまま使うのならもうほとんど出来上がったも同然だ。Apple が提供しているドキュメントは以下のアドレスにあるが、けっこう長いので読むのにめげる人もいるかもしれない。もちろん、いろいろな考え方は重要ではるのだけど、端的にどうすればいいかということだけを説明すると、MOSA のセミナーのときのようにシンプルに終わってしまうのである。それだけ、Cocoa がうまく作られていることに尽きるのであるが、いちおう今回からの一連の記事ではすでに説明したことも含めて、細部まで丁寧に解説を行う予定である。

◇Cocoa: Programming Topic: Document Based Applications
http://devworld.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/Documents/index.html

しかし、ここでも「プログラミング」と名売っているけど、実際のところ、コアな部分だけを構築するには、いわゆるコードをちまちま打ち込む作業はほんとに少ない（MOSA の資料をお持ちの方は御覧いただけるはずだ）。というか、動くソースを打ち込むだけならほんとうに少ししかプログラムは出てこない。もちろん、実用的なアプリケーションを作るにはそこから大量のコーディングは必要になるとは言え、もはやフレームワークベースの開発ではソースコードよりも別のところに払うエネルギーの方が多いことも意味しているだろう。そういうわけで、プログラミングだけど、要は Interface Builder での設定などの方が、ある意味では重要なポイントになってきてい

る点は意識していただきたいところだ。

なお、MOSA の講義で、ちょっとうまく行かなかった Revert ができなかったことと、画像が埋め込めなかった点は、理由が分かったので、それも織りまぜて説明をするつもりだ（NSTextView のアトリビュートに設定するチェックボックスの問題であった）。

(2)Application Kit の機能

Cocoa フレームワークの大きな特徴である、文書ファイルを伴うアプリケーションを作成する機能についての解説にさっそく入ろう。文書ファイル、たとえばテキストエディタとか、画像編集ソフトのように、アプリケーションとは別にデータをファイルに保存し、そのファイルをウインドウに開いて編集するという形式のアプリケーションは、もはや定番というのもおかしなくらい、パソコンでは当たり前になっている。そうしたアプリケーションを作るためには、オブジェクト指向以前（その意味では現在の Carbon も含む）の API では、ウインドウを画面に出すだけでもけっこうなプログラミングが必要であった。10 年くらい前のマックのプログラミング本だと、延々とプログラミングを行って、やっとシンプルなテキストエディタができました…という感じであった。

しかしながら、オブジェクト指向が一般的になったことで、差分だけを与えるというやり方がうまく機能するようになった。つまり、どのアプリケーションにも共通するような機能については、あらかじめフレームワークで作り込んでおく。たとえば、ウインドウを表示し、ウインドウの大きさを変えたりウインドウの位置を変更するといった動作は、ワープロでも画像ソフトでもほぼ共通だ。そうした機能はすでにフレームワークで用意しましょうというわけである。そして、ウインドウに表示する中身は、アプリケーションごとに作って下さいという具合である。もちろん、その中身を作る上でもオブジェクト指向によって部品が容易に使えてさらにはカスタマイズもできるなど、フレームワークの機能はふんだんに利用できる。そして、ウインドウといったコンポーネントだけでなく、「文書ファイルを扱うアプリケーション」という機能もフレームワークに含まれることが多い。いくらか複雑な仕組みになってしまうのであるが、それでもこうした機能を使えば、従来のやり方に比べて遥かに容易にアプリケーションができてしまうのである。なお、PowerPlant でも文書ファイルを扱うクラスは存在している。一方、Swing などの Pure Java では明確には存在していない。その意味では、Mac OS X のネイティブ環境である Cocoa で対応があるということは、Cocoa の魅力であると言ってもいいだろう。ちなみに、Cocoa ではこうした種類のアプリケーションを「Document Based Application」と呼んでいるので、以下「ドキュメントベースのアプリケーション」などと呼ぶことにする。

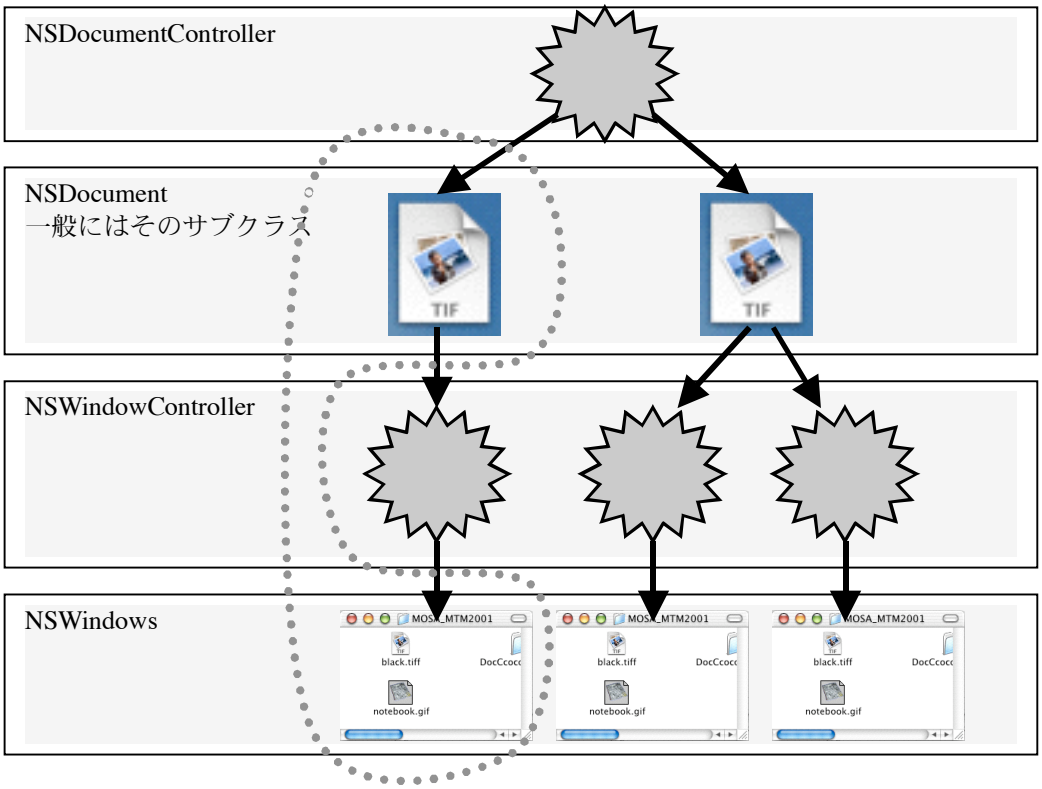
Cocoa でドキュメントベースのアプリケーションを実現するために、いくつかのクラスが用意されているが、その中心となるのは NSDocument である。最低限、このクラスのサブクラスを定義するということが必要になる。それ以外のクラスについては、基本的には特にいじらなくてもとりあえずアプリケーションは作成できる。しかしながら、フレームワークで定義されている機能を変更したりあるいは追加したいとなると、そうしたクラスを自分で作り直したりといったこともできるようになっている。今回は、NSDocument クラスだけを詳しく説明するが、その他のクラスは名前と動作の概要だけはとりあえず理解しておきたいところだ。

この NSDocument を利用する上では、nib ファイルの動作を理解しておきたい。nib ファイルは Interface Builder で作成できるもので、そこには、ウインドウやその中のボタン、メニューといったユーザインタフェース要素、そしてクラスの定義やそのインスタンスの定義、インスタンス間でのリンクといった参照関係の定義など、さまざまな情報が記録される。通常、アプリケーションは、NSApplication というクラスで実現されており、そのクラスをインスタンス化するところからアプリケーションは始まる。もっとも、main.m という Objective-C で書かれたソースが最初からプロジェクトに組み込まれているので、あまり意識はしたことがないかもしれない（この部分の解説は、Objective-C の NSApplication のクラスの部分で見ることができる）。しかしながら、そこでは、プロジェクトの「アプリケーション設定」にある「メイン nib ファイル」に指定した nib ファイルを自動的にロードするといった動作を行う。通常は、ここも最初から用意されている nib ファイルである MainMenu が設定されているので、MainMenu.nib に設定されたメニューがアプリケーションのメニューとして表示されるという動作になるわけだ。

一方、ドキュメントベースのアプリケーションの場合、必ずしもそのやり方しかできないわけではないものの、一般にはドキュメントの定義に対応した nib ファイルを定義する。nib ファイルは、ファイルとして存在するときには「クラス」という意味合いが強いが、それがロードされることによって「インスタンス化」されると考えればよい。だから、アプリケーションで使う nib ファイルとは別に「文書ファイル」を定義する nib ファイルを用意しておく。通常は、アプリケーションの中でドキュメントを複数開くことが多い。したがって、文書を開くたびに、その文書ファイルに対応した nib ファイルを利用してプログラム上で利用するインスタンスを生成するという方法が、いちばん素直なやり方なのである。

Application クラスにあるドキュメントを関するクラスをまとめておこう。動作から考えると、文書ファイルがいくつかあり、さらに 1 つの文書ファイルで複数のウインドウということもあり得るというあたりを想像してもらいたい。そうした状況を実現できるように Cocoa は作られている。以下の図は、クラス階層ではなく、実行した段階でのインスタンスの階層であるので注意してもらいたい。

ドキュメントを管理するクラス



Nib ファイルで定義する

まず、いちばん根底に NSDocumentController というクラスがある。これはマルチドキュメントをサポートするクラスで、アプリケーションでは 1 つだけインスタンス化されている。このクラスはけっこういろいろな仕事をこなすが、何もしなくてもインスタンス化されていて使える状態にあると考えて良い。ここでは、文書を新規に作ったり、文書を開いたり、さらには終了時に閉じるかどうかを訪ねるとか、Finder でのドラッグ&ドロップとの連動といったことまでも行う。また、そうした動作を変更したいときには、このサブクラスを定義して利用することもできてしまう。今回のサンプルはとりあえずそのまま利用することにする。

次に NSDocument クラスがあり、1 つの文書ファイルに 1 つのインスタンスを用意する。ただし、文書のデータ処理に関しては、やはりどうしても共通に動作するものは無理なので､実際に作成するアプリケーションに合わせた部分を作らないといけない。そのため、通常はサブクラスを作ってそれを利用する。この動作については、一連の原稿で詳しく紹介しよう。

さらに、NSWindowController クラスがある。1 つの文書ファイルに対して 1 つ用意し、1 つの文書に対するマルチウインドウをサポートする。これも、基本となるクラスは定義されていて、通常は NSDocument ないしはそのサブクラスを作ると自動的に用意されるものと考えて良い。ただし、サブクラスを作ってカスタマイズすることも可能となっている。

そして、NSWindow クラスがあるが、これはウインドウそのものである。1 つのウィンドウに対して、1 つのインスタンスを使うことになるが、もちろんこれもサブクラスでもかまわない。単に通常のウインドウの動作をさせて、その中にコントロール類があるだけなら、通常はそのままを使う。また、言い換えれば、この NSWindows クラスは nib ファイルで定義しているものをそのまま使うのが一般的である。プロジェクトの初期状態がそうなっているので、まずはそうした使い方からマスターするのがいいだろう。つまり、nib ファイルで作成したユーザインタフェースに基づくウインドウを作成するという具合である。

このようにいくつかのクラスがあって、それらが連係を取りながら動いて行く。しかも、自動的にいろいろ行われてしまうので、結果オーライとするのなら何も説明することはなくなってしまうのであるが、なるべく背後でのフレームワークでの動きを筋道を立てて説明していきたいと考える。

(3)作成するアプリケーションの仕様

前回までに、Cocoa のフレームワークには NSDocument を中心にして、文書ファイルを使ったアプリケーション作成を支援する機能があることを説明し、その機能の概要や概念的なことを説明した。いよいよ実際に作りはじめるわけだが、いちおう、目標を立てることにしたい。とりあえずは、MOSA のセミナーで紹介したということで、アプリケーションは「MOSAEditor」とうい名前にしよう。まずは第一段階は「基本的なテキストエディタ機能」を実現するということにする。次のレベルを目指すことにする。

- ウインドウを開いて、そこにテキスト編集のコントロール（NSTextView）を配置する。もちろんそこではキータイプや各種の編集が可能であるが、コントロール自体はそのまま使う
- 作成した文書を文書ファイルとして保存できる。また、文書ファイルを開くことができる
- アプリケーションや文書にアイコンをつけ、文書ファイルをダブルクリックなどすれば、開くことができる

以上のことが実現した後は、概ね次のような予定で、機能をアップして行くことにしたい。（MOSA のセミナーのときと順序が違っている。）また、コピー&ペーストやドラッグ&ドロップ、サービスメニュー対応などは、そのままの設定でどこまでサポートされているのかということがポイントになるが順次説明を加える。

- 復帰（Revert）をサポートする
- リッチテキスト対応エディタにアップグレードする。フォント設定の機能などを付加して、ファイルフォーマットもそれに対応する
- 初期設定機能を組み込む。メニューのアクティベートやユーザインタフェースの作成などがポイントとなる
- ローカライズする。日本語のリソースの準備と現実的な手法を解説する

文書フォーマットを考えておくことにしよう。上記のシナリオだと、テキストファイ

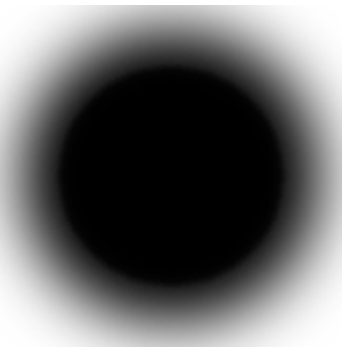
ルと Ritch Text の 2 通りが出てくる。テキストファイルの場合の文字コードはシフト JIS としてファイルの拡張子は、一般的に.txt とする。一方、Ritch Text ファイルの拡張子は.rtf でもいいのだが、他のものと当たらない medit ということにしよう。アプリケーションのクリエイタは「ome9」としておく。

それから、アイコンのファイルを作っておこう。いろいろな手順があるし、ツールもある。Developer Tools には Icon Composer もあるが、ちょっと使いにくいというのもあるので、やはり何かツールはないのかと考えるところだ。ちょうど、MOSA の Software Meeting で快技庵の高橋さんが紹介していた Icongrapher というのがよさそうだったので、これを使ってみることにする。$15 のシェアウエアである。

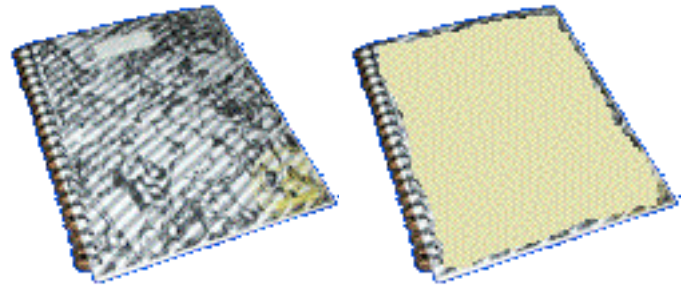
◇Icongrapher 日本語版
http://www.mscape.com/products/iconographerJ.html

元の画像として以下のものを用意した。Icongrapher でもある程度は編集作業はできるのであるが、ここでは Photoshop や Graphic Converter を使ってある程度は作ってしまったものを使うことにする。mycat.jpg がアプリケーションアイコンで、そのマスクとして 8 ビットのグラデーション画像を black.tiff として用意した。そして、テキストファイル用に notebook2.gif、medit ファイル用に notebook1.gif を使うことにする。いずれも、128 ドット四方の最大サイズのアイコン画像を作った。なお、デザイン的に考慮されたものではない点はご容赦いただきたい。

Mycat.jpg と black.tiff

notebook1.gif と notebook2.gif



Icongrapher を使って必要なアイコンをまとめたファイルを作る方法を説明しよう。最終的には icns ファイルに保存するが、その中身はいろいろなサイズのアイコンやマスク画像が集合体となったものだ。もちろん、Icongrapher を使えばそうしたファイルフォーマットは気にしなくてもいいのだが、いろいろな形式の画像は基本的には必要になる。ただ、ここでは最大サイズの画像を用意して、それを単に縮小するということでアイコンを用意するという簡単な方法を取ることにする。

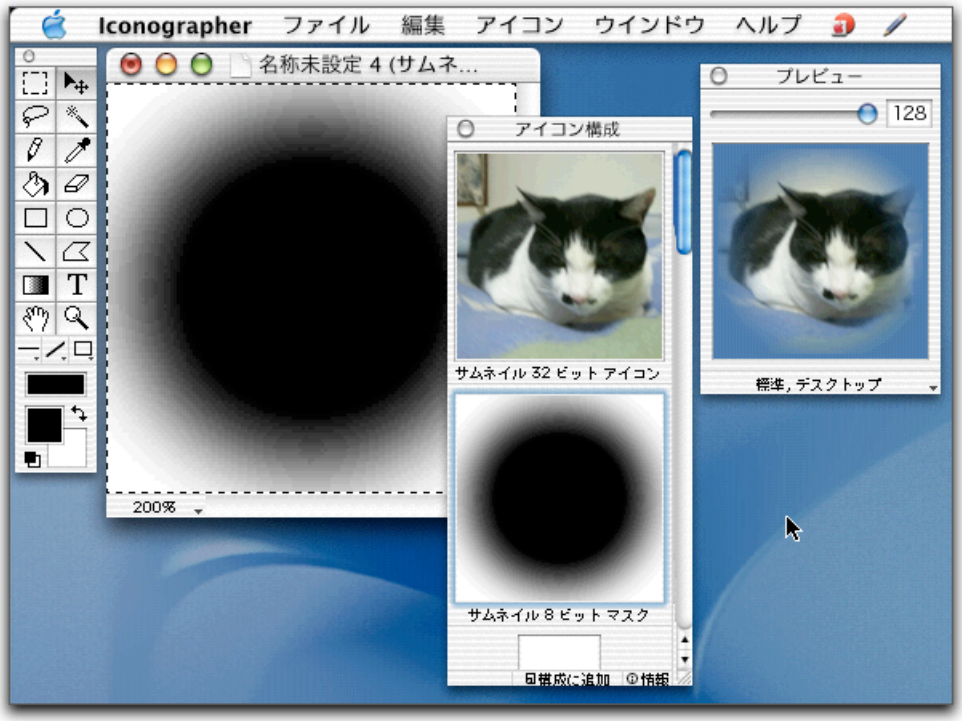
まず、Icongrapher で、「ファイル」メニューの「新規アイコン」（Command+N）を選択する。右側で「アイコン構成」というパレットがあるので、その中にある最大サイズの「サムネイル 32 ビットアイコン」を選択する。最初は真っ白のはずだが、選択することでブルーの枠で囲われる。そして、別のアプリケーションでコピーした画像を、ここでペーストする。たとえば、mycat.jpg ファイルを Graphic Converter で開き範囲選択して Command+C でコピーする。そして、Icongrapher のサムネイルを選択いて Commap+V でペーストするというわけである。

サムネイルの画像をペーストした

ところで、このペーストを行うとき、画像を表示するウインドウの倍率を 200%ほどにして、ウインドウ内に隠れる部分がない状態で行わないと正しく画像がペーストされなかった。

続いて、同じように、マスク用の画像 black.tiff を開いてコピーし、それをアイコン構成パレットの「サムネイル 8 ビットマスク」を選択してペーストを行った。プレビューのパレットで、写真とマスクが合成されているところがすでに参照できるようになっている。

サムネールのマスクをペーストした



そして、「アイコン」メニューの「アイコンの作成」を選択すると、アイコン構成パレットのその他のアイコンにすべて縮小したり減色したアイコンあるいはマスクが設定される。もちろん、デザインにこだわる場合には 1 つずつ作成しないといけないのであるが、おおざっぱでいい場合はこうした手軽なアイコン作成が可能である。

残りのアイコンを自動的に生成した

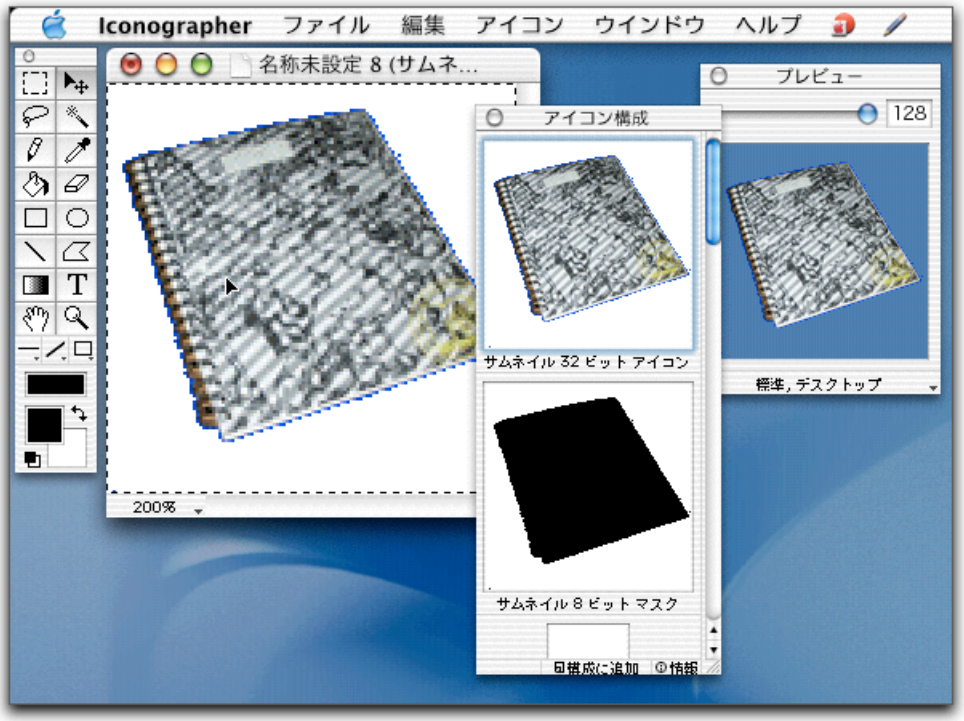
アイコン画像が全部できれば「ファイル」メニューから「保存」（Command+S）を選

択するなどして、ファイルを保存する。このとき、Mac OS X 向けのアプリケーションなら、フォーマットして「Mac OS X(.icns ファイル)」を選択しておけばよいだろう。ファイル名は適当に付けるが、拡張子は icns にしておくのがよいだろう。

アイコンファイルとして保存しておく

続いて、同じように、notebook1.gif をベースにしたアイコンを作成するが、もともとの画像の周辺部分を白で塗りつぶしておく。そして、アイコン構成パレットのサムネイル 32 ビットアイコンにペーストするが、その画像をすぐ下のサムネール 8 ビットマスクにドラッグ&ドロップすると、白い部分を抜くようなマスクを自動的に作成してくれる。

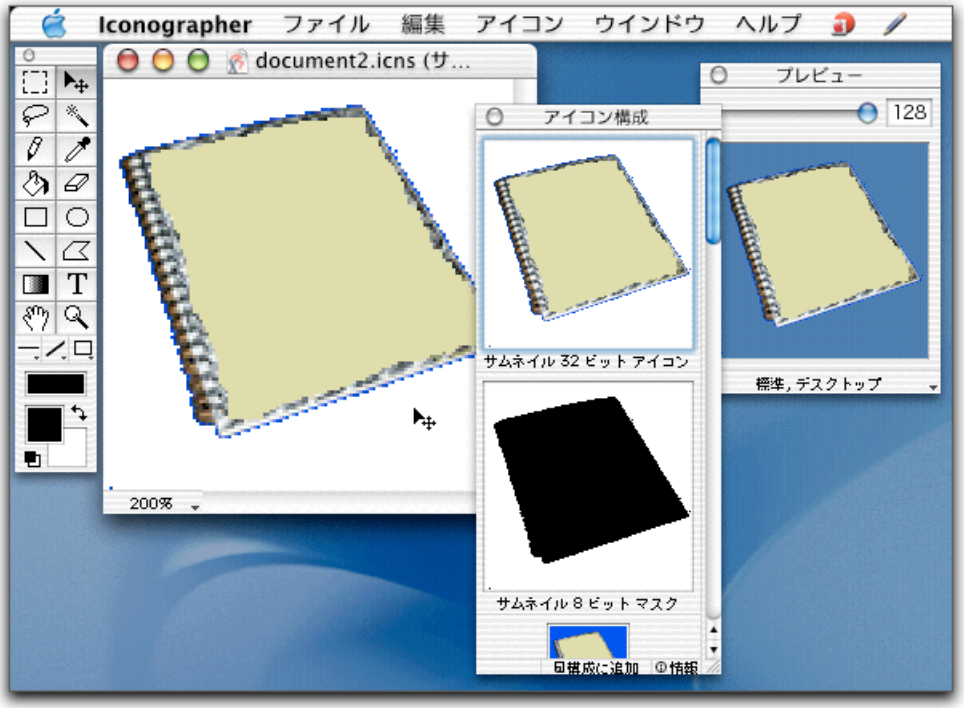
マスクは画像をドラッグ&ドロップして作成した



なお、この notebook1.gif を元にした画像の場合、なぜか「アイコンの作成」が機能してくれないので、他の領域にも、元画像をドラッグ&ドロップして手作業で各構成のアイコンをうめて行かなければならなかった。このファイルは、document1.icns として保存した。

さらに、notebook2.gif をもとにしたアイコンファイル document2.icns も作成した。こちらはなぜか「アイコンの作成」はきちんとすべての構成をうめてくれた。

もう 1 つのアイコンファイルも作成した。



これらの作成されたアイコンファイルをダウンロードするのであれば、以下のリンクをご利用いただきたい。

http://mdonline.jp/figs/01/0047/application.icns
http://mdonline.jp/figs/01/0047/document1.icns
http://mdonline.jp/figs/01/0047/document2.icns

筆者がまだ Icongrapher になれていないせいもあるのかもしれないが、今回は操作方法でいろいろはまってしまった…。できれば、透過 GIF を読み込んで、自動的にマスク設定するとかいった機能や、アイコン構成パレットの部分に Finder からファイルをドラッグ&ドロップして登録するといった機能があると便利ではないかと思った次第である。

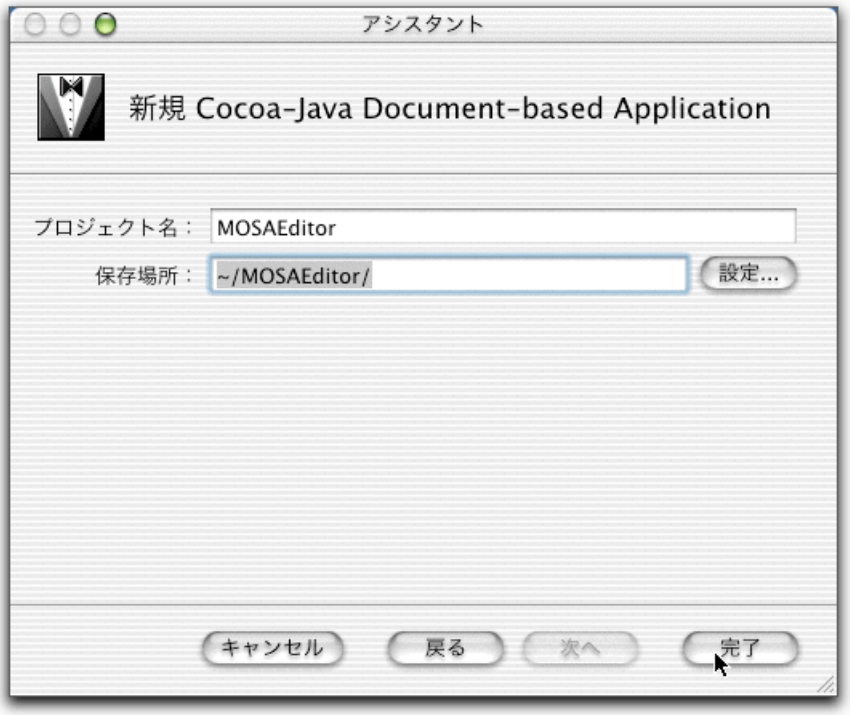
(4)プロジェクトの用意

今回から実際に Deloper Tools を使ってプログラミングに入ろう。もっとも、プログラミングとは言ってもコードを書くのはほんのちょっとだけであるが…。仕様するツールは、December 2001 版の Developer Tools である。

まずは、Project Builder でプロジェクトを作成する。「ファイル」メニューから「新規プロジェクト」（Command+shift+N）を選択し、ダイアログボックスで「Cocoa-Java Document-based Application」を選択する。そして、プロジェクト名とそのフォルダを保存するフォルダを指定する。今回の一連のプログラムは MOSAEditor ということにしよう。

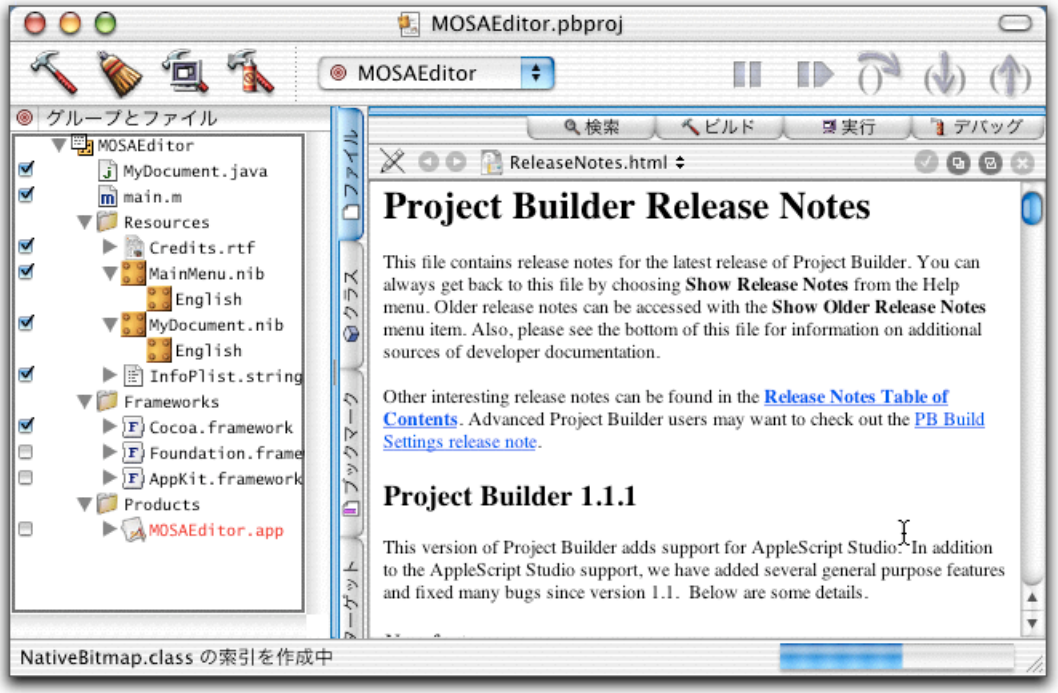
Cocoa-Java Document-based Application を選択する

プロジェクト名と保存フォルダを指定する



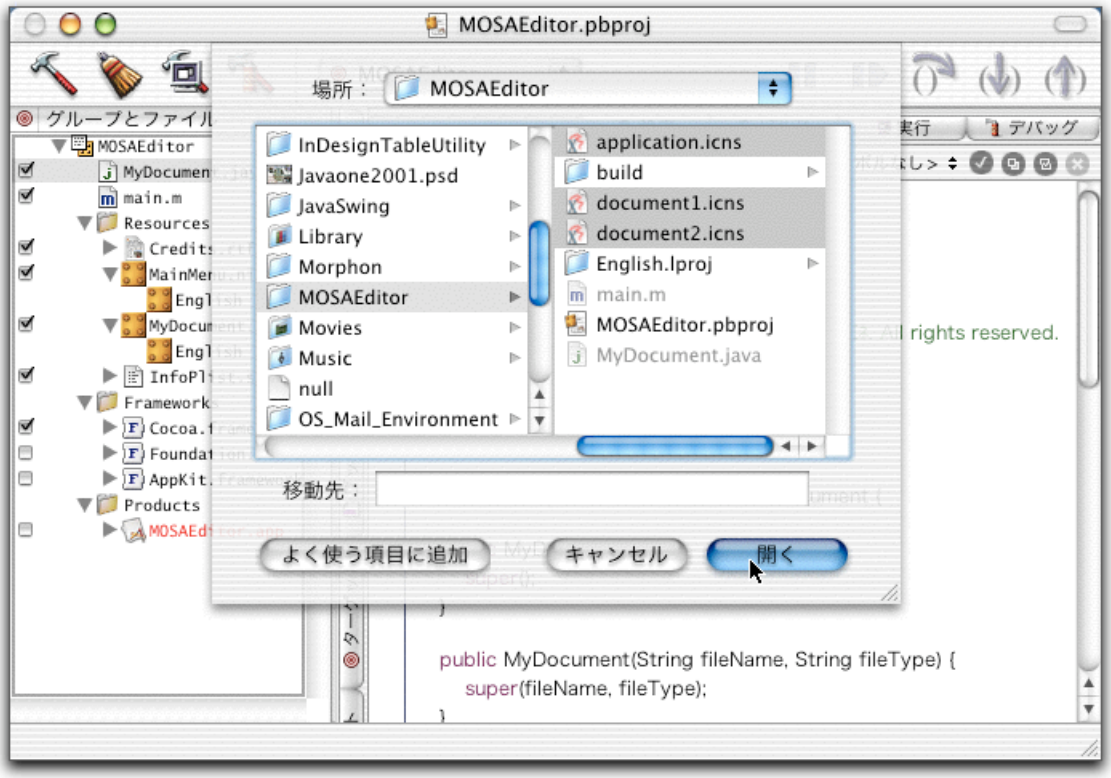
作成されたプロジェクトを見て見よう。まず、アプリケーションで必ず必要になる main.m があるが、これはそのままで OK である点は他のアプリケーションと変わらない。そして、nib ファイルが 2 つあるところが目に付く。ここではまず、MainMenu.nib があるのだが、それに加えて、MyDocument.nib という nib ファイルもある。そして、Java のソースファイルとして、MyDocument.java というものもある。まず、これらの基本となるファイル群をチェックしておこう。

作成したばかりの Cocoa-Java Document-based Application のプロジェクト

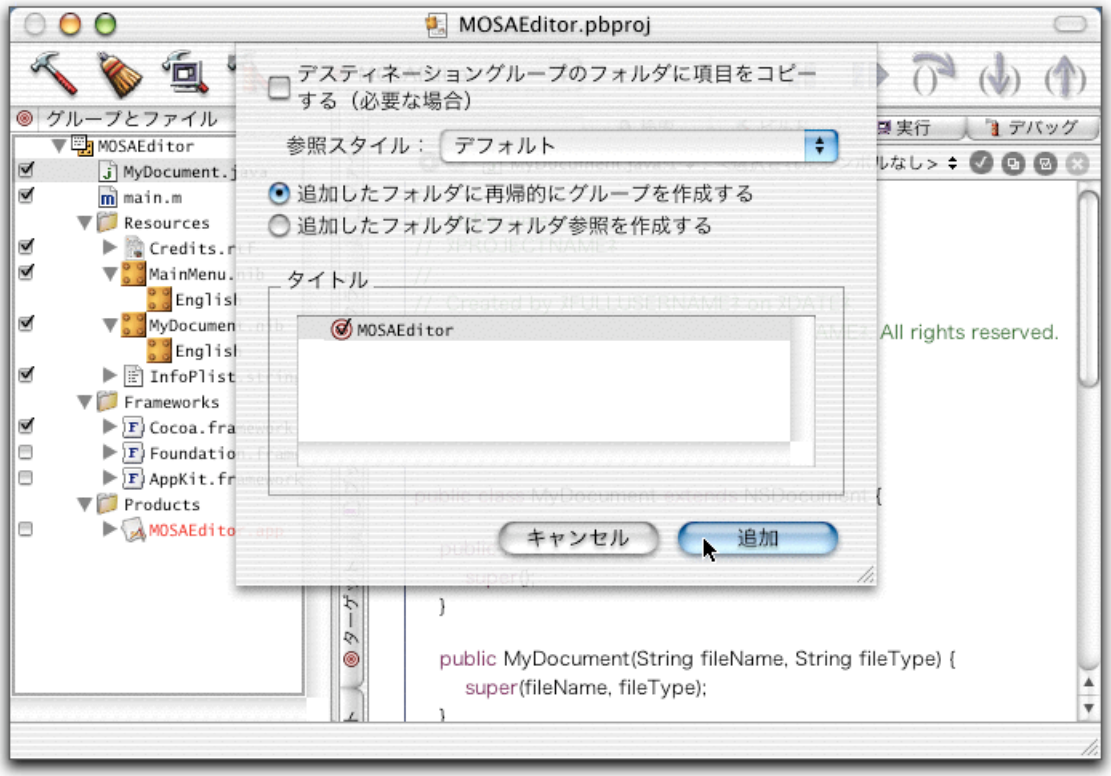


必要なファイルを先にプロジェクトに登録しておこう。前回、アイコンファイルを 3 つ作ったが、そのファイルを、プロジェクトのフォルダに Finder でコピーしておく。そして「プロジェクト」メニューの「ファイルを追加」（Command+option+A）を選択して表示されるダイアログボックスで、3 アイコンファイルをまとめて選択して、「開く」ボタンをクリックすれば良い。その後に、登録するターゲットなどの設定シートが表示されるが、プロジェクトの中のファイルの場合はデフォルトで問題はないだろう。

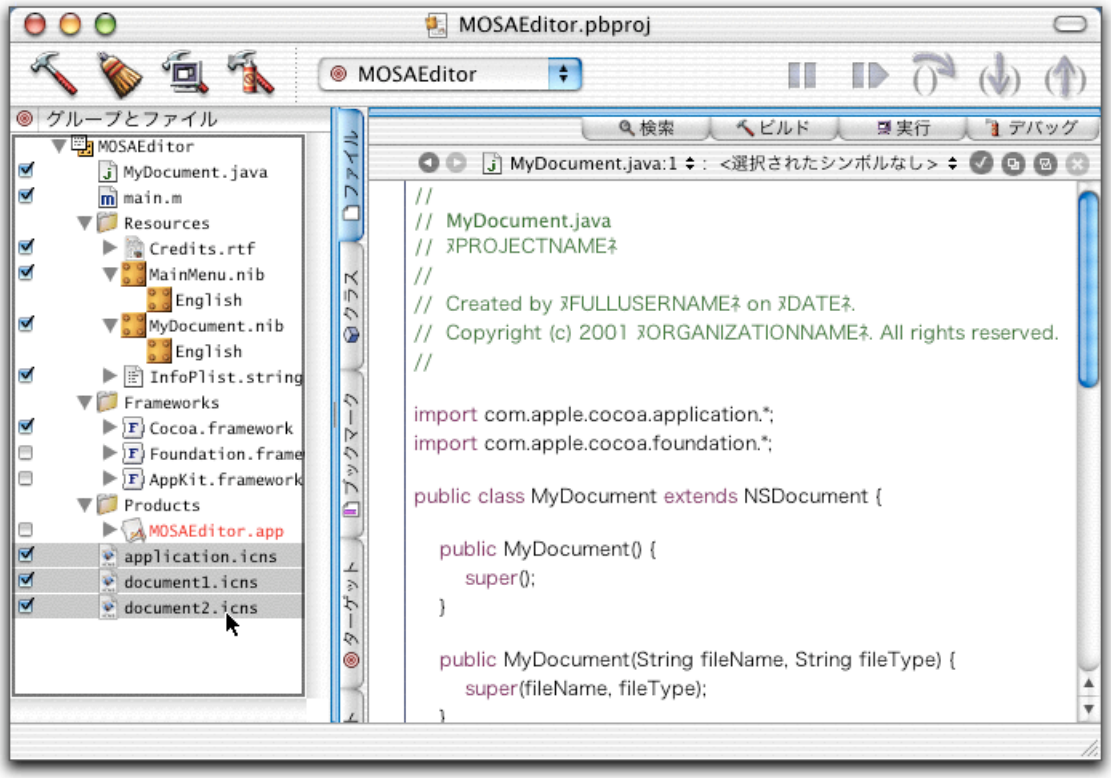
アイコンファイルをプロジェクトに追加する



ファイルの追加方法はそのままで OK



アイコンのファイルがプロジェクトに追加された



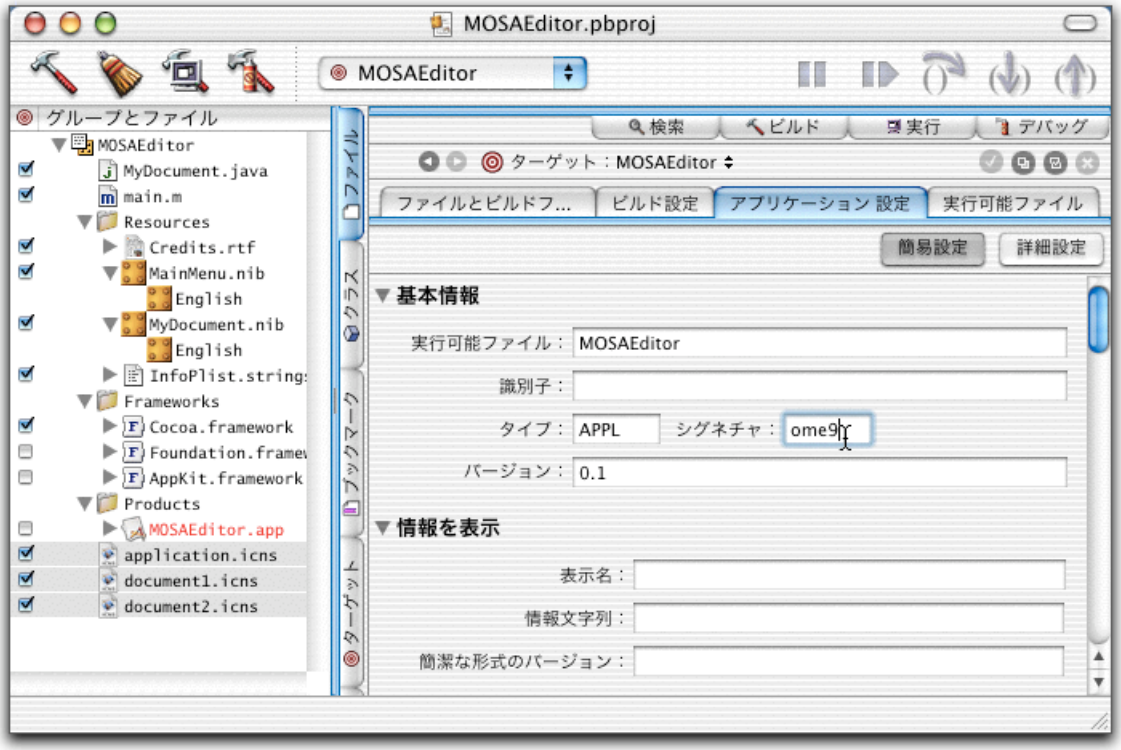
ファイルがプロジェクトに追加されると、左側のリストに出てくる。必要ならグループを作るなどして分類していいだろう。ファイル名の左にあるチェックボックスは、現在のターゲット（ターゲットはビルド方法に関する 1 つの設定）で利用されるかどうかを示している。通常はチェックが最初から入っているのだが、新しいターゲットを作ったときなどはこのチェックボックスで、必要なものや不要なものを振り分けるということができる。

とりあえず必要なファイルを組み込むことができたので、次は、アプリケーションに必要な設定を行うことにしよう。「プロジェクト」メニューから「アクティブターゲットの編集」（Command+shift+E）を選択して、ターゲットの編集パネルを表示する。そこにある「アプリケーション設定」を選択する。

まずは「基本情報」を見て見よう。「実行可能ファイル」はプロジェクト名がそのままなので、これはそれでいいだろう。識別子は特に指定は必要ない。タイプとシグネチャは設定の必要がある。シグネチャ（クリエイタ）は、ここでは ome9 に書き換えるとする。Mac OS X でもシステムはシグネチャを使ってアプリケーションを特定するという対応付けを行っている。もちろん、シグネチャは Apple に申請して唯一のも

のを取得する必要がある。バージョンは任意に設定すればいい。

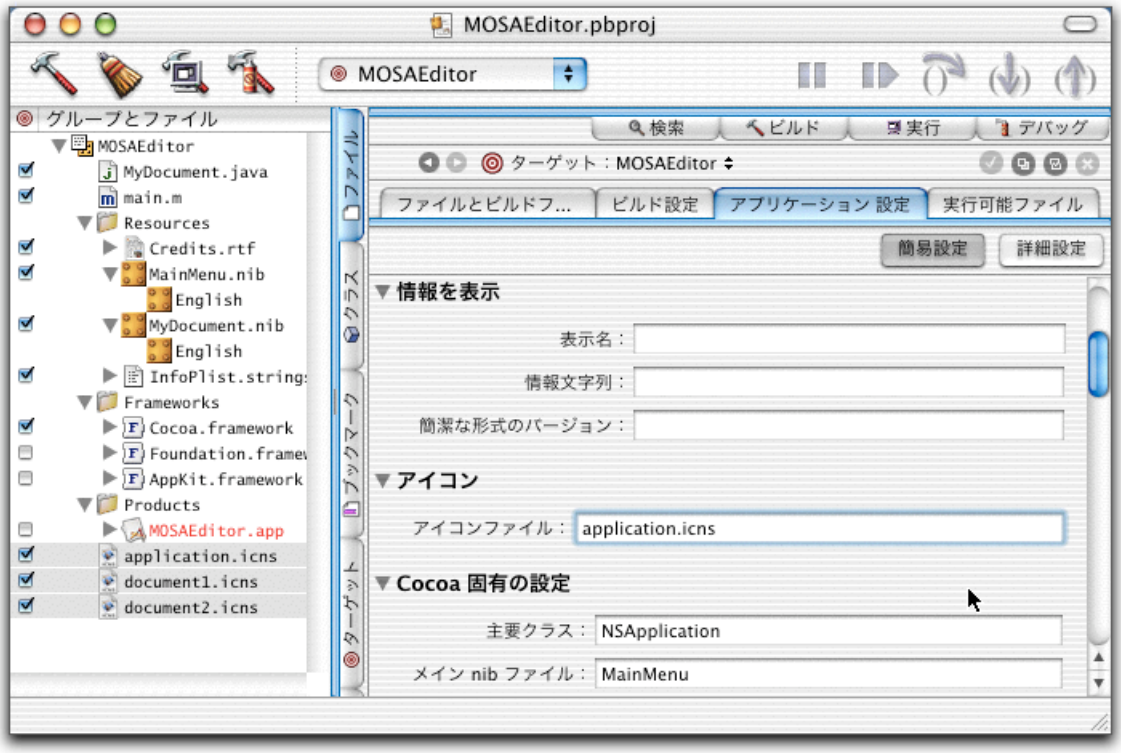
アプリケーション設定の基本情報を設定する



続いて、「アイコン」の部分ではアプリケーションのアイコンファイルのファイル名をキータイプして入力する。さきほど追加したアイコンファイルのファイル名をここに入力すれば良い。

「Cocoa 固有の設定」は最初からすでに記入されているが、ここが重要である。主要クラスが NSAplication、メイン nib ファイルが MainMenu となっている。この「メイン nib ファイル」という設定は、このアプリケーションを起動したときに自動的にロードされる nib ファイルを指定するのである。つまり、MainMenu.nib ファイルに定義したオブジェクトは、起動した段階で自動的に展開されるということである。実際にここではメニューなどを定義しているが、メニューがこうしたメカニズムで自動的にインスタンスとなってアプリケーションの中で利用できると考えておけばよいだろう。

アプリケーション設定のアイコンと Cocoa 特有の設定

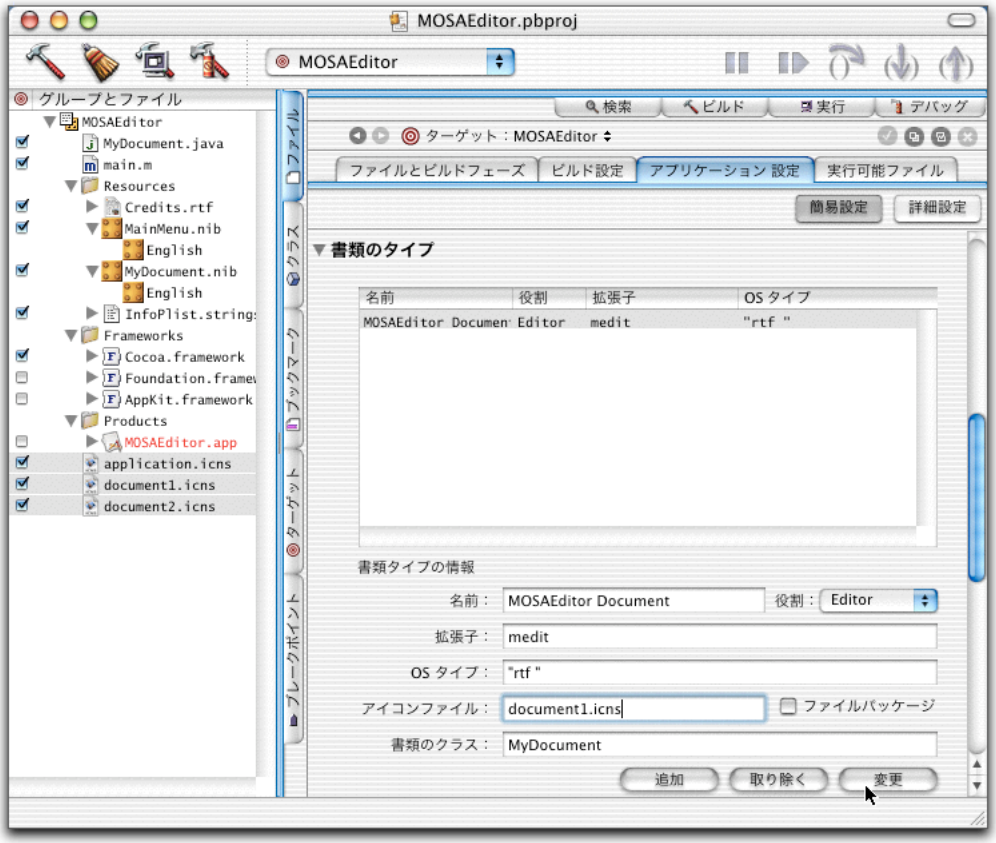


続いて「書類のタイプ」という部分の設定を行う。ここでは、文書ファイルの定義に従った設定を行うが、すでに 1 つの設定が組み込まれている。まず、この設定を書き直すのであるが、一覧の中の設定をクリックして選択すると、下側の部分に設定内容が表示される。ここで書き直して「変更」ボタンをクリックすれば良い。

まずは、リッチテキストフォーマットの文書を保存する設定を定義する。書類タイプの情報の「名前」は書類の名前を指定するが、これは Finder の文書情報の種類に出てくる文字であることを意識しておく必要がある。「役割」は Editor でいいだろう。続いて「拡張子」はあらかじめ決めておいた medit を指定する。「OS タイプ」は、ファイルタイプであるが、ここで指定したファイルタイプも同様なフォーマットであるとして読み込み可能にすると考えればよい（これだとリッチテキストのファイルタイプとしては不適格かもしれないが…）。ファイルタイプとの兼ね合いなどは実際にプログラムを加えてファイル処理ができるようになってから改めてチェックしよう。「アイコンファイル」はもちろん用意したファイルのファイル名を指定する。「書類のクラス」は最初から MyDocument が設定されているがこれが非常に重要な意味を持つ。すでに作られている MyDocument.java ファイルで定義されたクラス名が指定されるが、ここでは medit 拡張子のファイルを開いたとき、そのドキュメントの管理を MyDocument クラスのインスタンスで行うということを意味する。つまり、ファイル

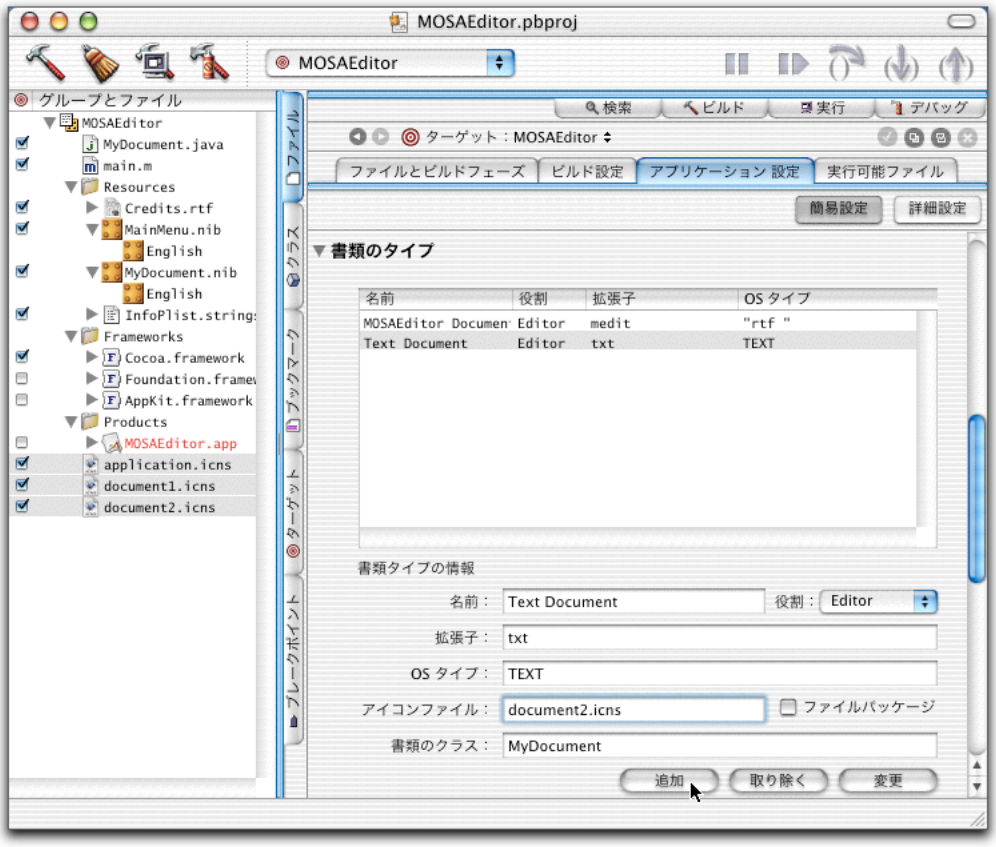
と処理を行うクラスの対応付けを、「書類のクラス」の設定で行うというわけだ。

medit 拡張子のファイルの文書ファイルとしての定義を行う



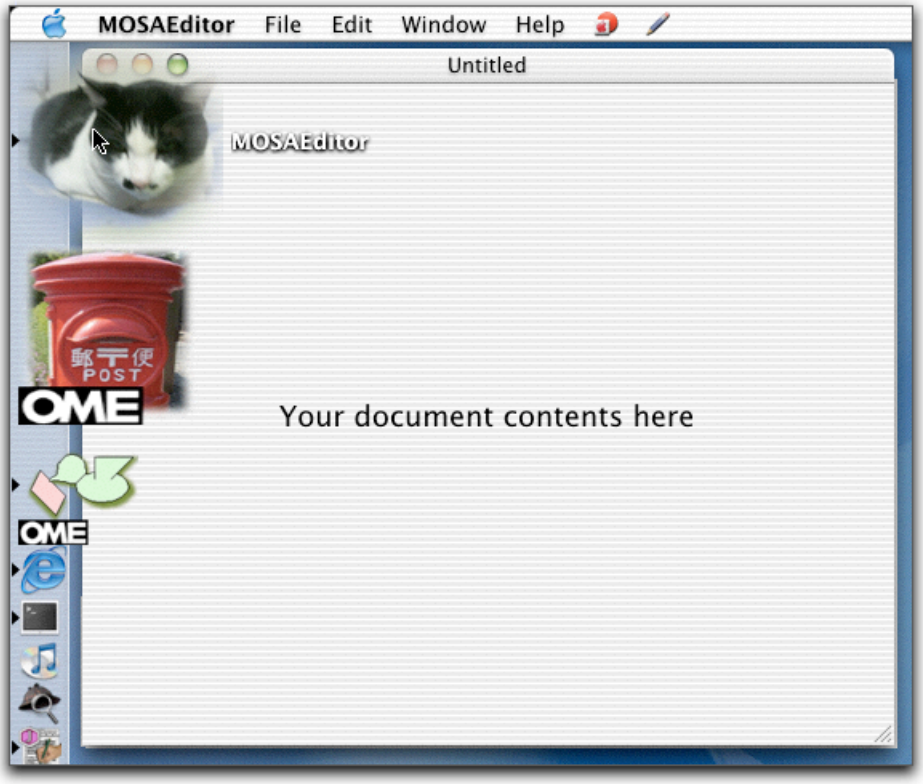
続いて、テキストファイルに関する定義も追加する。その場合は、リストの下側にある領域に必要な項目をとにかく記入して、「追加」ボタンをクリックすれば良い。ここでは「OS タイプ」にはテキストファイルのタイプである TEXT をしっかり記載しておく必要があるだろう。

テキストファイルの文書ファイルとしての定義を行う



まだ、nib ファイルなどを確認はしていないものの、このままとりあえず実行してみよう。プログラムを 1 行も追加していないので、まずは問題なくコンパイルして実行できるはずだ。実行はツールバーの左から 3 つ目のボタンをクリックするか、Command+R で行う。すると、次のようにとにかく MOSAEditor が起動した。アプリケーションアイコンはさっそく、作成したアイコンファイルが見えている。メニューは英語であるがとにかく存在する。また、ウインドウが 1 つ開いている。Your Document Contents Here とあるのだが、とりあえずは何もできない。

プログラムに変更を加えない状態で起動してみた。メニューやウインドウがすでに出ている

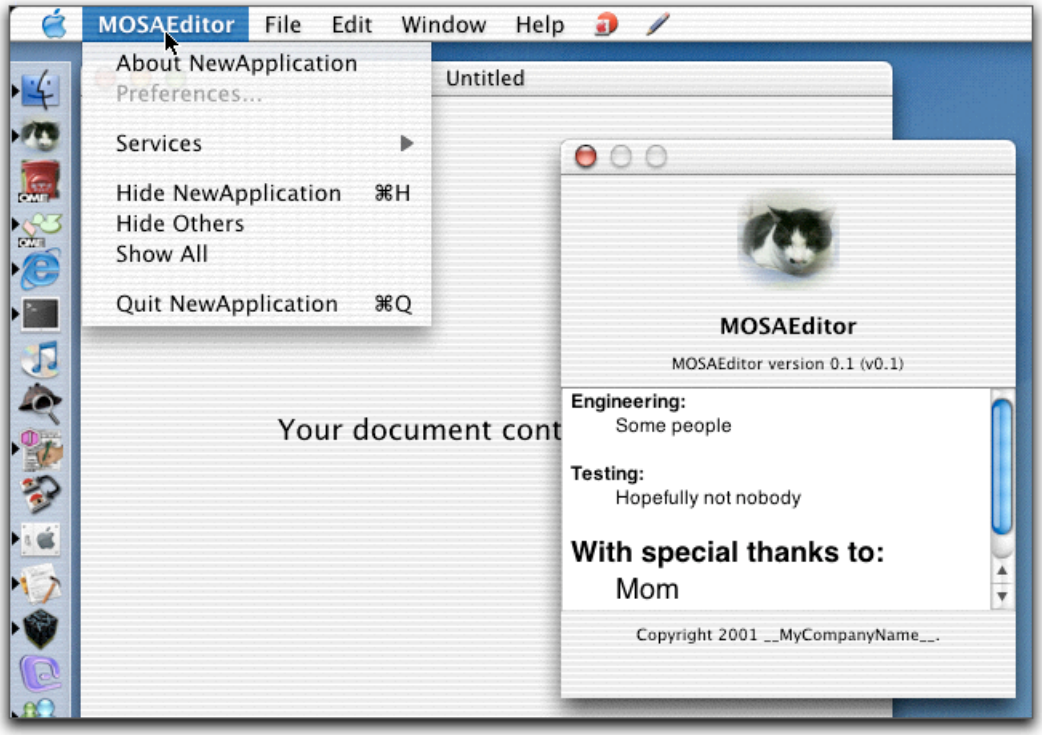


実際の設定との兼ね合いは改めて説明するが、メニューについては、MainMenu.nib に定義されたものがそのままアプリケーションで利用されている。ウインドウについては、MyDocument.nib で定義されたものが表示されているのである。これらは当然ながらカスタマイズしないといけないのだが、その方法は次回以降で説明しよう。

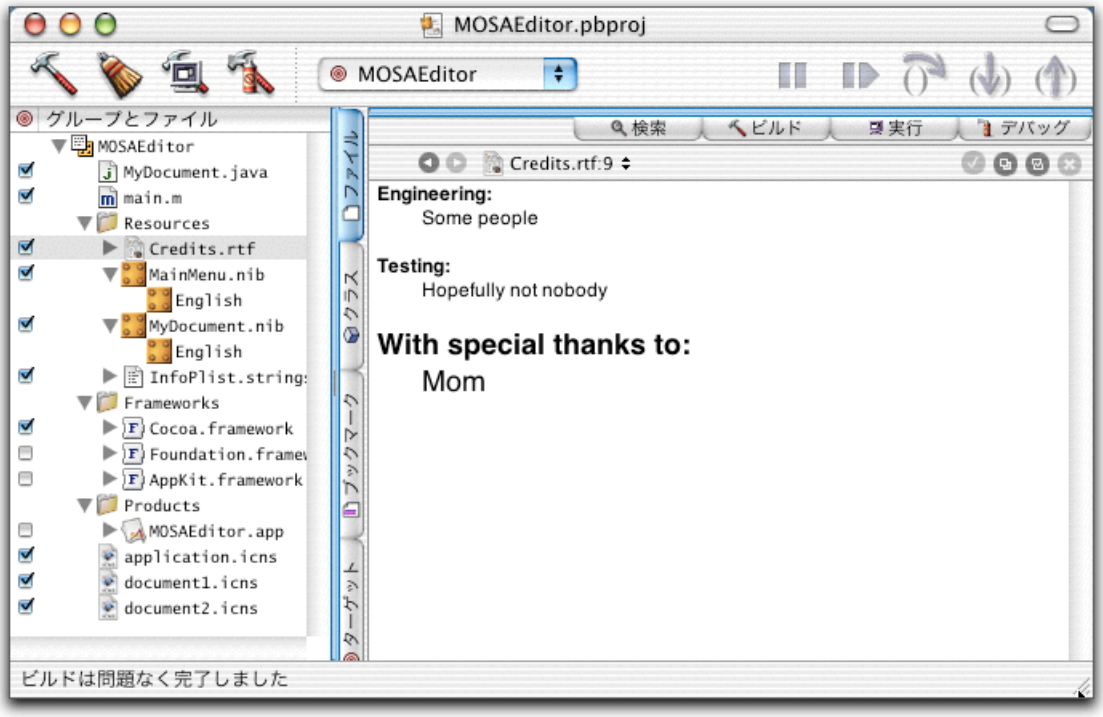
(5)初期状態での動きを見る

まずはテンプレートのままの状態で動いているアプリケーションの動作を追って行く。実は、何のプログラミングをしなくてもここまで動くというのがフレームワークの強力なところなのである。アプリケーションメニューを見てみよう。英語ではあるが、終了もできるし、隠すこともできる。一部のメニュー項目が NewApplication になっているが、これは後から変更しないといけない。ここで、About NewApplication を選択すると、すでにデフォルトのアバウトウインドウが表示する。この内容は、プロジェクトファイルにある Credits.rtf の中身である。このように、RTF ファイルを用意すれば簡単なアバウトウインドウまでもサポートしてしまっているのである。

アプリケーションメニューは既に使える状態にある

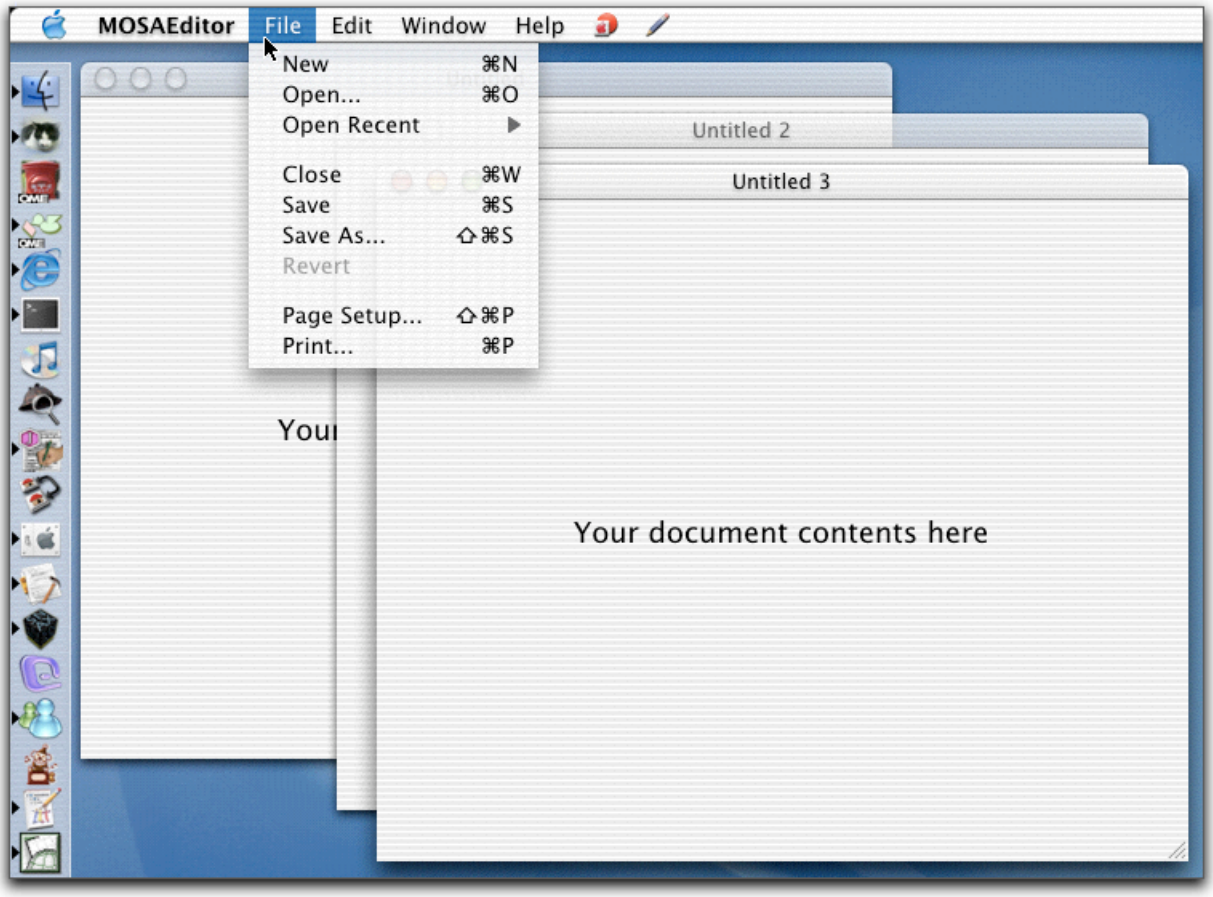


アバウト画面のメッセージは RTF ファイルで与えることができる



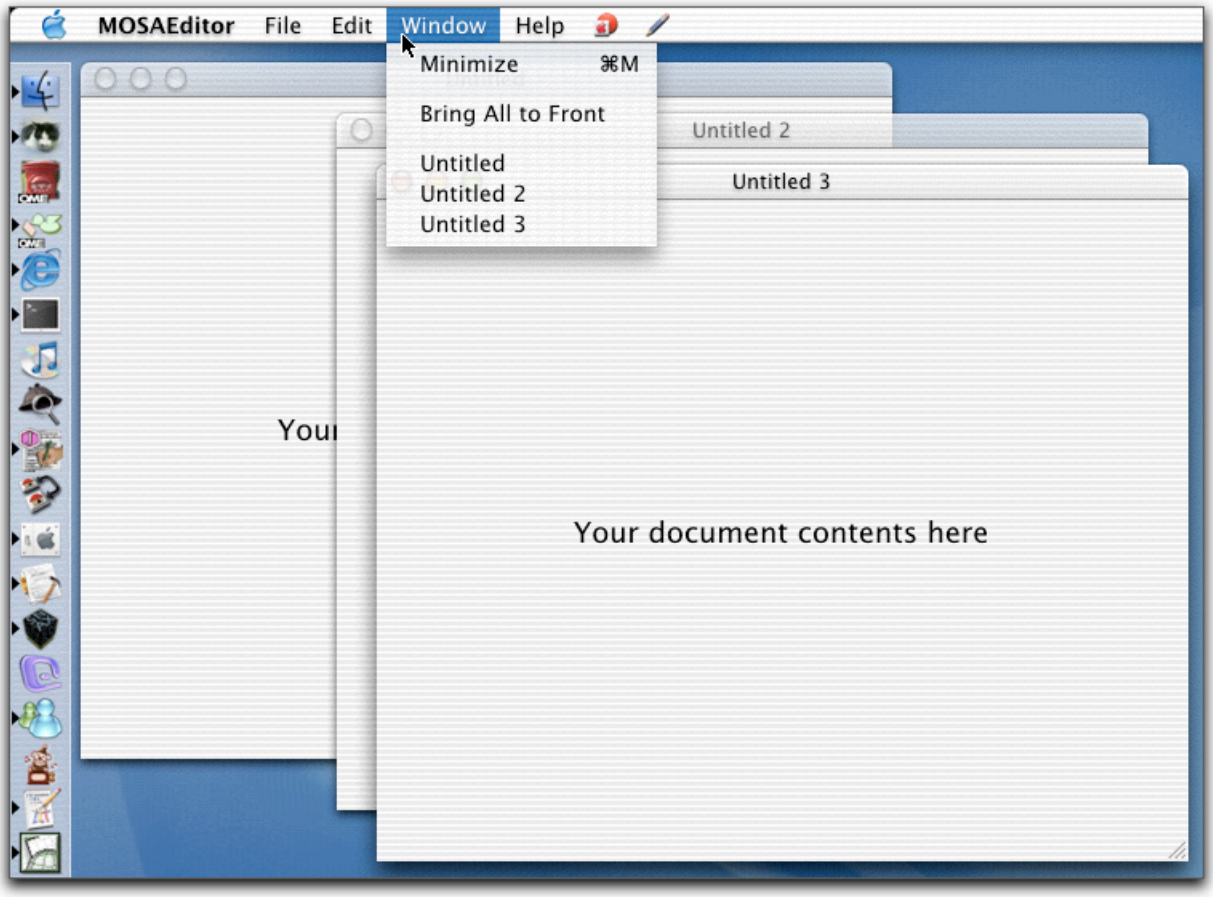
File メニューを見てみよう。こちらも、ほぼ、必要なメニュー項目が揃っているのが分かるが、ここで、New を選択してもらいたい。もちろん、Command+N でもかまわない。ショートカットも何もしなくても機能している。

File メニュー



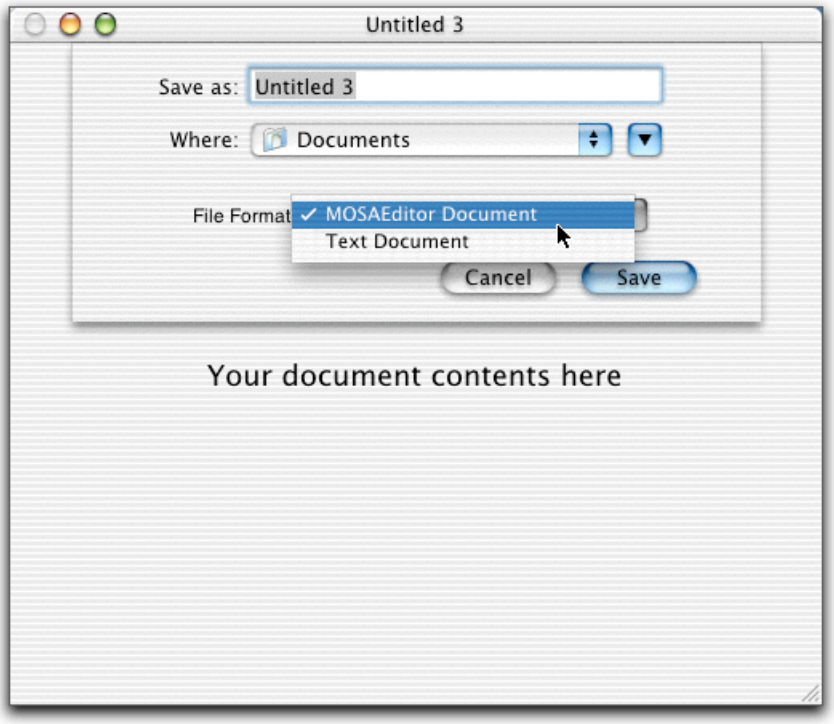
New を選択すると、新たにウインドウがいくつも作られる。ウインドウをいくつも作って Window メニューを見てもらいたい。すると、すでに作られたウインドウの項目が見えている。また、File メニューの Close（Command+W）も機能している。これら、基本的なマルチウインドウ処理は、何もしなくてもアプリケーションに組み込まれてしまっている。もちろん、ウインドウのタイトルバーをドラッグして移動したり、サイズを変更するという処理も、特に何もしなくても組み込まれているのである。

Window メニューのメンテナンスも自動的に行われている

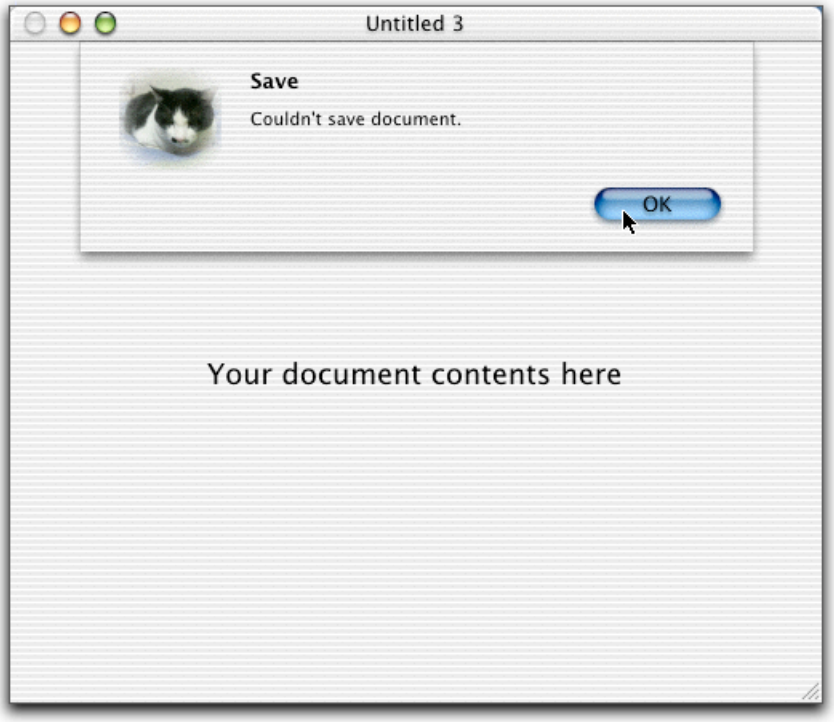


ここで、File メニューから Save（Command+S）を選択してみよう。すると、保存するファイルとフォルダを指定するシートが出てくる。File Format を注目してもらいたいが、ここで、すでにアプリケーション設定の「書類のタイプ」で指定したタイプの名前がポップアップに並んでいる。実際に保存しようとしても、エラーメッセージが出る。これは正常な動作である。保存に際しては、データの取り出しを自分でプログラムしないといけないのだが、初期状態ではそうしたプログラムは含まれていないし、エラーが出るようになっているというわけだ。

ファイルへの保存をやってみる。すでに書類タイプを認識している



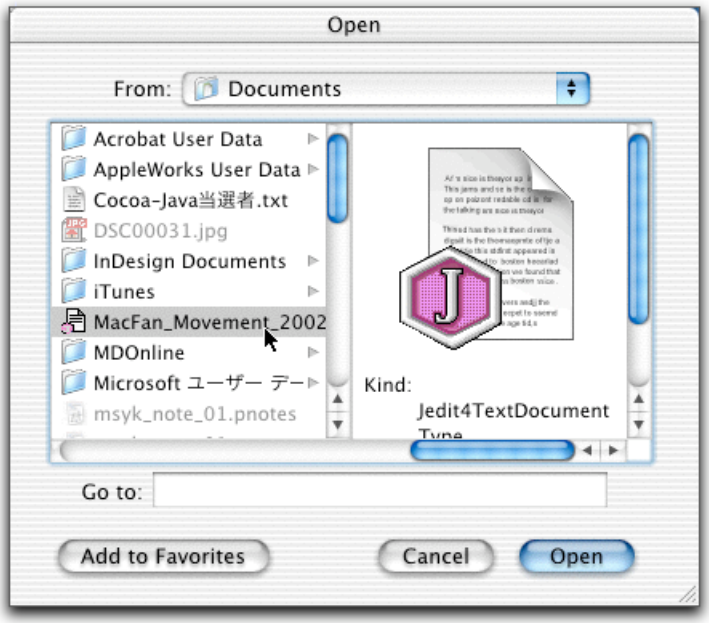
保存しようとしたがエラーになる。これは正常な動作だ



今度は、File メニューから Open（Command+O）を選択してみよう。ダイアログボックスが表示されるが、ここでもやはり書類タイプの認識が行われている。たとえば図に見えている拡張子が.txt のテキストファイルは TextEdit によるもので、ファイルタ

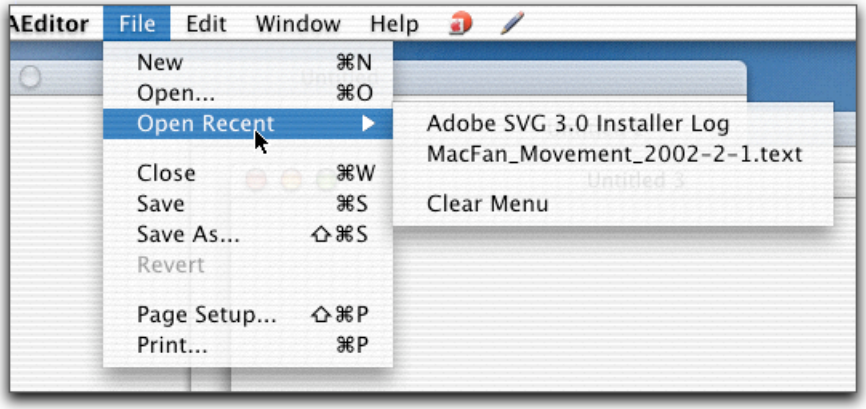
イプは未設定だ。一方の Jedit4 書類のテキストは拡張子は text なのだがファイルタイプは TEXT である。つまり、書類タイプで指定した拡張子やファイルタイプのあるものはグレーにならずに選択できるようになっており、定義に含まれないファイルはグレーとなって選択できないのである。ここでも書類タイプの認識をきちんと行っている。

ファイルを開く場合でもすでに書類の種類の認識が行われている



ここでファイルを開いても、ウインドウには何も出てこないが、ウインドウのタイトルだけは開いた書類のアイコンやファイル名が記載される。こうしてつらつらといくつかのファイルを開くと、File メニューの Open Recent に開いたファイル名が記録されていく。そして、選択すれば再度開くといったことが行われる。この Open Recent（最近開いた書類）についても、実はフレームワークですべてメンテナンスしてくれるので、プログラマは何もしなくても（というのは大げさだけど）この機能が使えるというわけである。

Open Recent の追加は自動的に行っている



以上のように、一切プログラミングを行わない状態でも、マルチウインドウ、マルチドキュメントのアプリケーションとして動いてしまうほど、Cocoa の中で機能が作り込まれている。後は、必要なメソッドの組み込みなどをすれば OK である。逆に言えば、どこまでが自動化されていて、どの部分を自分で組み込む必要があるのかを知るのが、Document ベースのアプリケーション作成のポイントであると言えるだろう。次回はユーザインタフェースを組み込むことにしよう。

(6)既存の nib ファイルの設定を見てみる

文書ファイルを伴うアプリケーションのフレームワークは、以下の「Document Based Applications」というところに解説がある。以前は NSDocument クラスにあった解説がこちらに移動してきているわけだ。今回の一連のシリーズでは、もちろん、ここに書かれていることを詳しく説明するということを行っているわけだが、ここで、CustomInfo.plist ファイルを作成するようにという説明がある。「The Document Types Info Property List」ところでそのファイルの作成フォーマットが記載されているが、この情報を利用して、文書ファイルの認識などを行うとされている。

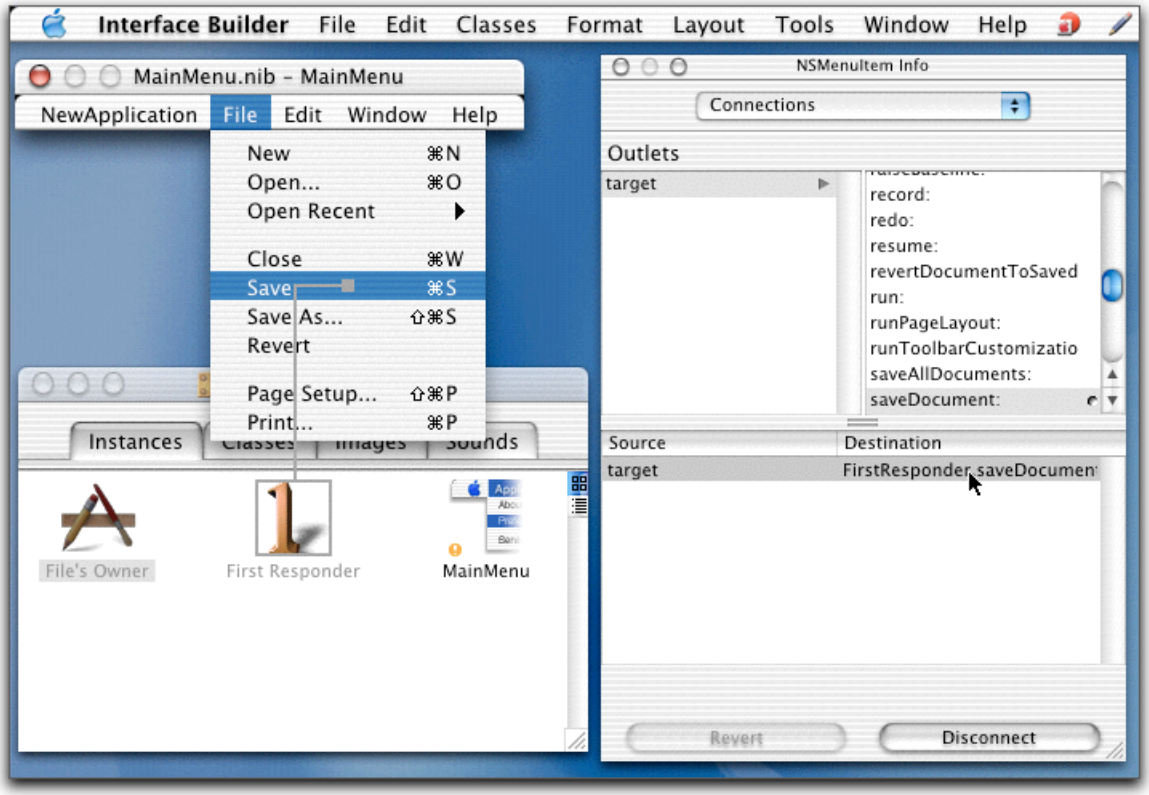
◇Document Based Applications
http://devworld.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/Documents/index.html

しかしながら、いろいろ試した結論から言えば「CustomInfo.plist は不要」ではないかと思われる。理由としては、まずプロジェクトの「ビルド設定」にある各種の設定と CustomInfo.plist の内容は重複している。つまり、CustomInfo.plist に与えるべき内容はビルドしたアプリケーションの Info.plist ファイルですべて記述されているはずのものなのである。また、Developer Tools には SimpleToolbar というサンプルアプリケーションがあるが、このサンプルでは CustomInfo.plist は作られていない。さらに、CustomInfo.plist を作らないで動かしても、前回に説明をしたように、特に問題は見られない。CustomInfo.plist はもしかして YellowBox 時代には必要だったものの、今現在はすでに不要となってしまった情報ファイルなのではないかとも思ってしまうがいかがなものだろうか？　つまり、ドキュメントの書き直しが行われていないという可能性が高いと考えられるのである。

それでは、Document-based Application のテンプレートで作られたプロジェクトにある nib ファイルを見ながら、テキストエディタに必要な設定を見て行くことにしよう。まずは、English の言語しかないが、ある程度作ってからローカライズということを考える。nib ファイルは 2 つあるが、MainMenu.nib は起動時にロードされるものだ。開いてい見ると、次の通り、メニューの定義「MainMenu」がある。そして、メニューの

各項目は、First Responder というインスタンスに結合されている。

MainMenu.nib の定義内容

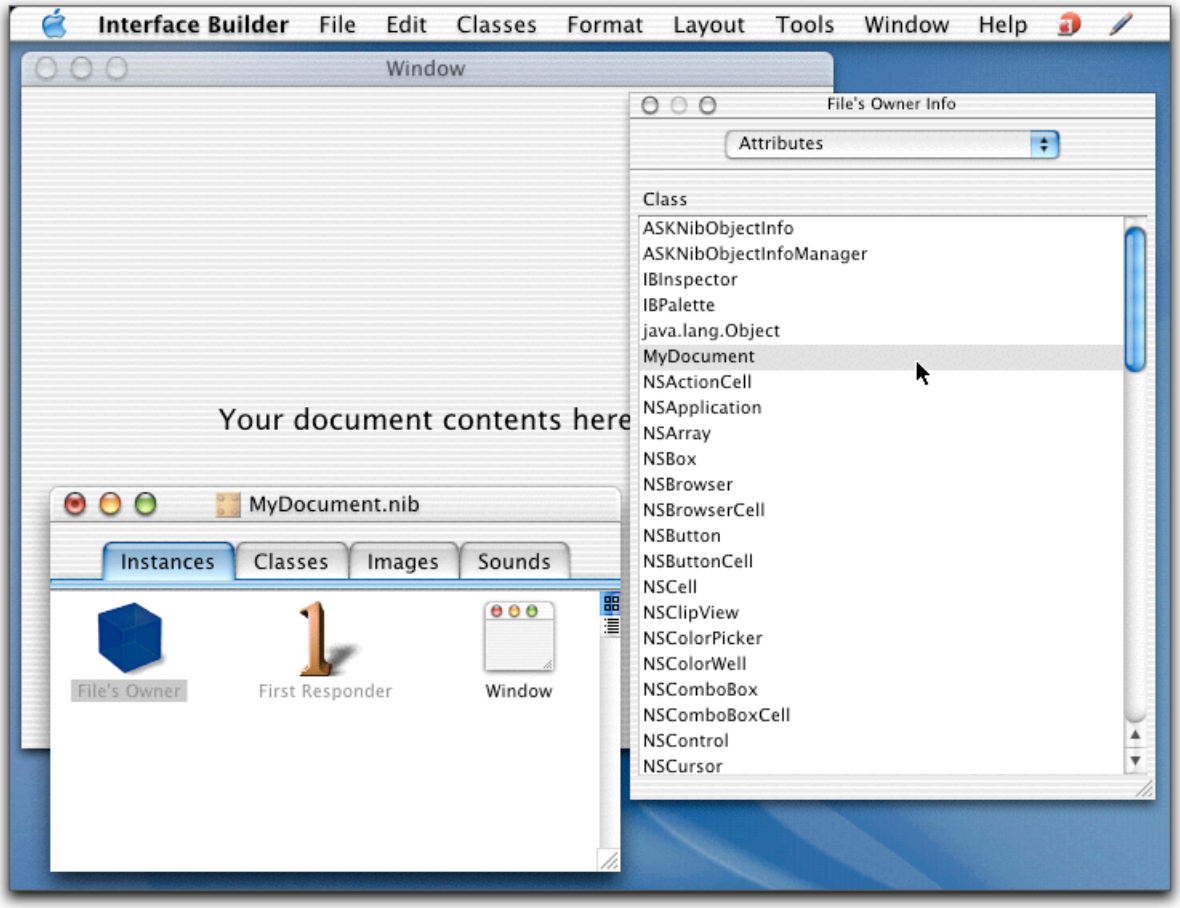


ここでの First Responder は、ユーザからの応答を受け付ける窓口と考えればよい。たとえば、Save メニューを選択すると、First Responder に saveDocument というメッセージが投げられる。First Responder は、現在のアプリケーションの状況を調べて、そのメッセージをさらに有効なインスタンスに引き渡す。たとえば、テキスト編集中だったら、テキスト編集のコンポーネントに渡すのであるが、たいがいはそこでは何もしないので、ビジュアルコンポーネントの階層に従って、ウインドウやドキュメントに伝達され、そこで saveDocument が用意されていて実際のファイルへの保存作業に移るといった仕掛けになっているわけだ。こうしたユーザから内部コンポーネントへのリクエストの伝達という点は複雑な仕掛けではあるが、Cocoa のさまざまな機能を使う上ではあまり意識しなくてとりあえずはうまく動くような状態になっている。なお、MainMenu.nib の File’s Owner はここではアプリケーション自体のインスタンスとなっている。とりあえず、MainMenu.nib は特に変更をしないで先に進むことにしよう。

次に、MyDocument.nib を見て見よう。これが文書ファイルを総合的に管理する仕組みを提供するのである。まず、File’s Owner についてチェックしたいが、Info パレットを表示して、Attributes をチェックしよう。つまり、このファイルの元になっているク

ラスがどのクラスなのかを示す設定であり、MyDocument クラスが選択されている。MyDocument クラスの実体は、ソースファイルの MyDocument.jara である。このあたり同じ名前で種類が違うものが並ぶが何がどこでつながりがあるのかをきちんと理解したいところだ。

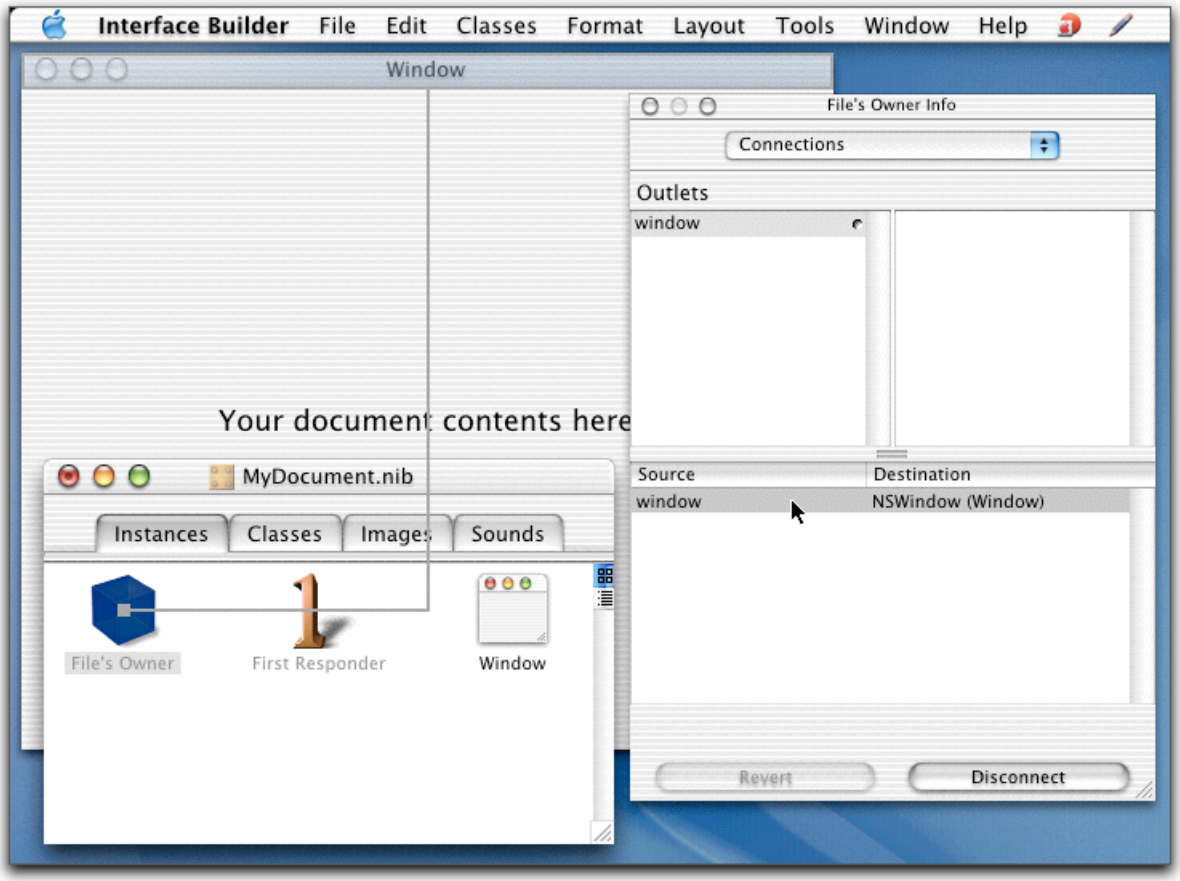
MyDocument.nib の定義内容



MyDocument.nib にある Window は、文書ファイルを表示するウインドウの実際のユーザインタフェースを定義しておくオブジェクトである。

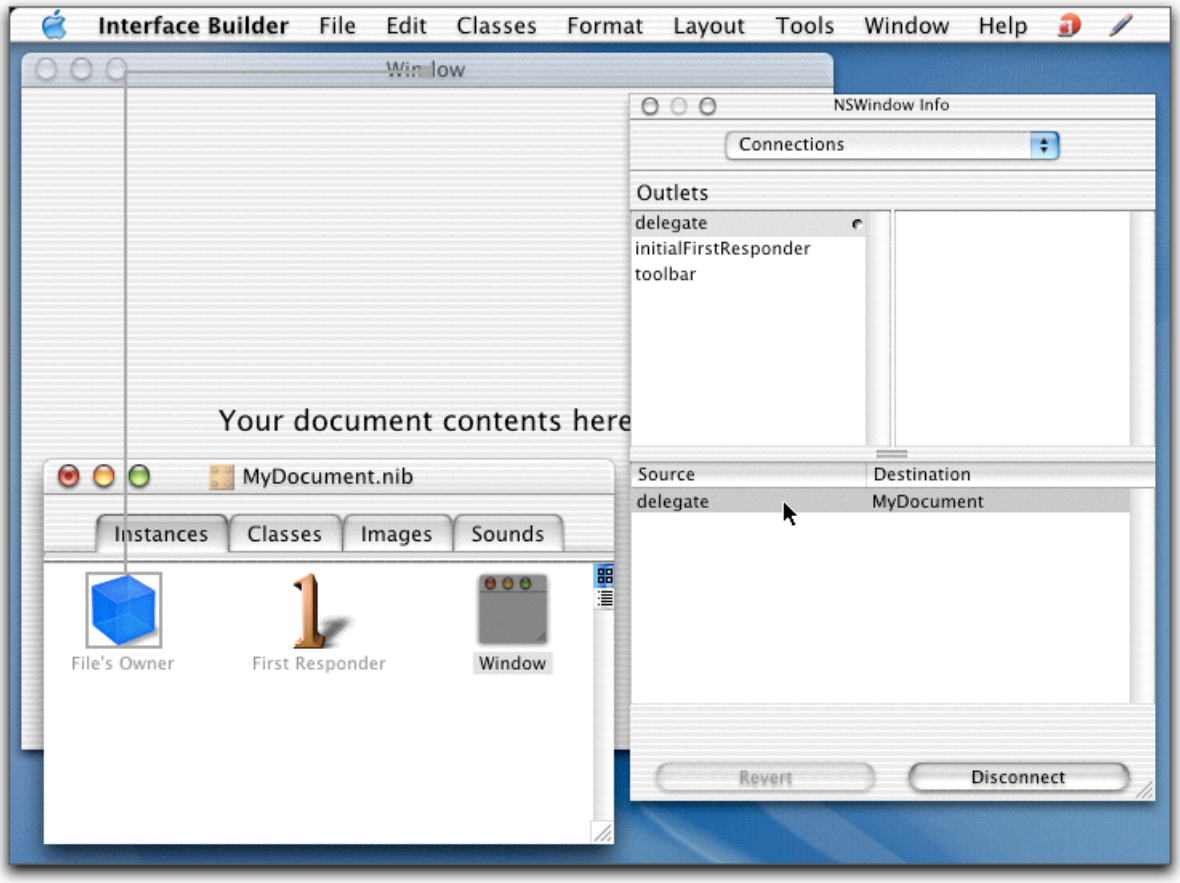
続いて、MyDocument.nib ファイルの初期設定を確認しておこう。まずは、File’s Owner を選択して、Info パレットで Connections を見ると、File’s Owner から Window にリンクした Outlet の window がある。つまり、ここでは、MyDocument.java というプログラムソースで、実際に表示されているウインドウを参照することを可能にする。今回のサンプルではここまでは使わないが、ウインドウの参照は細かな処理を組み込むと必ず必要になる。そのときは、すでに Outlet が定義されていることを理解しておく必要がある。

File’s Owner から Window を参照する Outlet が定義されている



一方、Window についても同様に Connections を見てみると、delegate という Outlet から、MyDocument つまり File’s Owner に対してリンクが設定されている。NSWindow クラスはデリゲートの処理が可能となっており、規定のイベントなどでデリゲートされた別のクラスのメソッドを呼び出すことができる。その規定のイベントに対応するメソッドを、ここでは MyDocument.java に記述することができるということだ。別に異なるインスタンスを用意してもかまわないのだが、その場合はリンクの設定を変更する必要がでてくる。このリンクもやはり今回のサンプルでは使わないとは言え、やはりより進んだ機能を組み込みたいときには、この初期設定があることを知っておいた方がいいだろう。

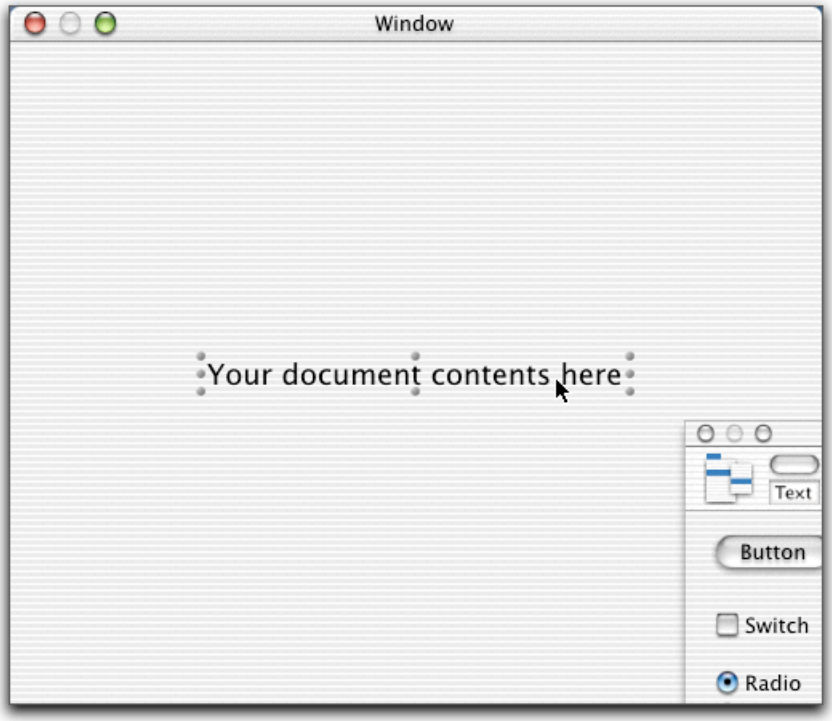
Window のデリゲートは MyDocument のインスタンスに設定



(7)文書ウインドウに NSTextView を配置する

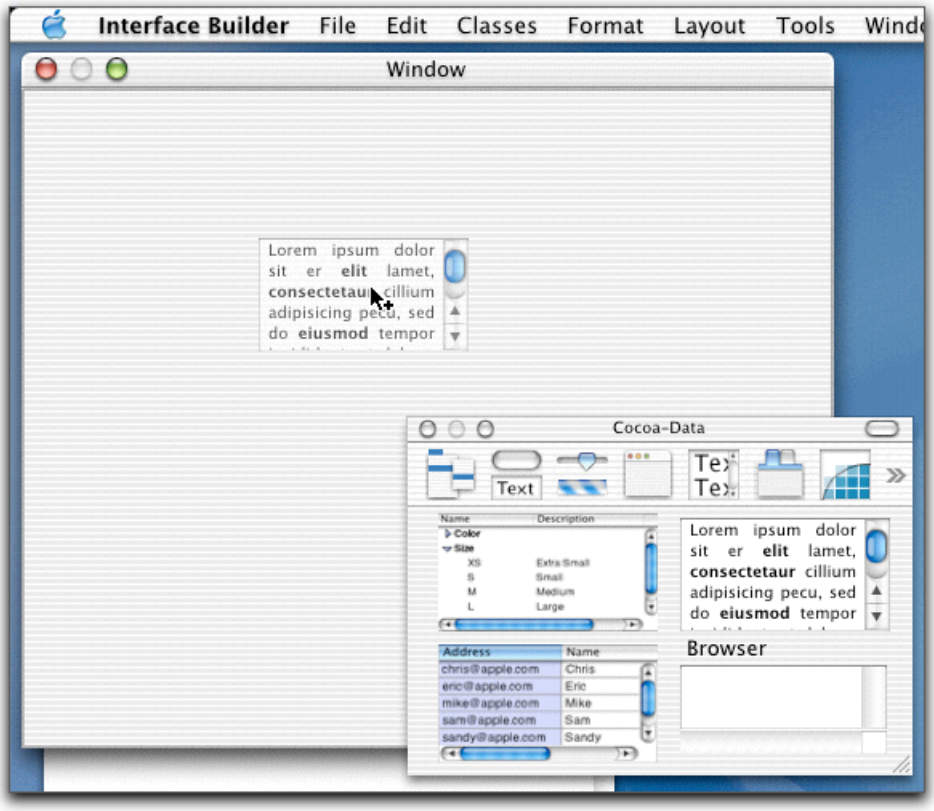
続いてウインドウの Windows にユーザインタフェースを組み込む。とは言っても、1つのコントロールを配置するだけだ。まずは最初から配置されているテキストは不要なので、これは削除する。クリックして選択して、delete キーを押せばそれでよい。

Window 内にある最初から組み込まれているテキストを選択して削除



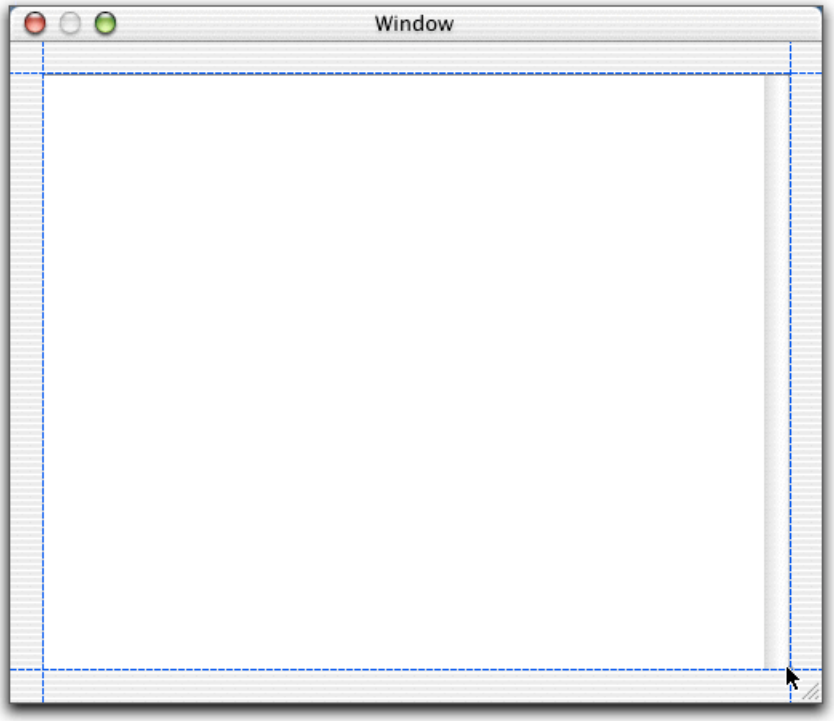
テキストエディタを簡単に作るには、NSTextView というコンポーネントを使う。ツールパレット上部では左から 5 つ目の「Cocoa Data Views」というアイコンを選択する。そして、右上の文字がいっぱい並んでいるボックスを、ウインドウの中にドラッグ&ドロップして、NSTextView を配置する。正確には、NSScrollView に包含された NSTextView というクラスのインスタンスである。

NSTextView をウインドウにドラッグ&ドロップして配置する



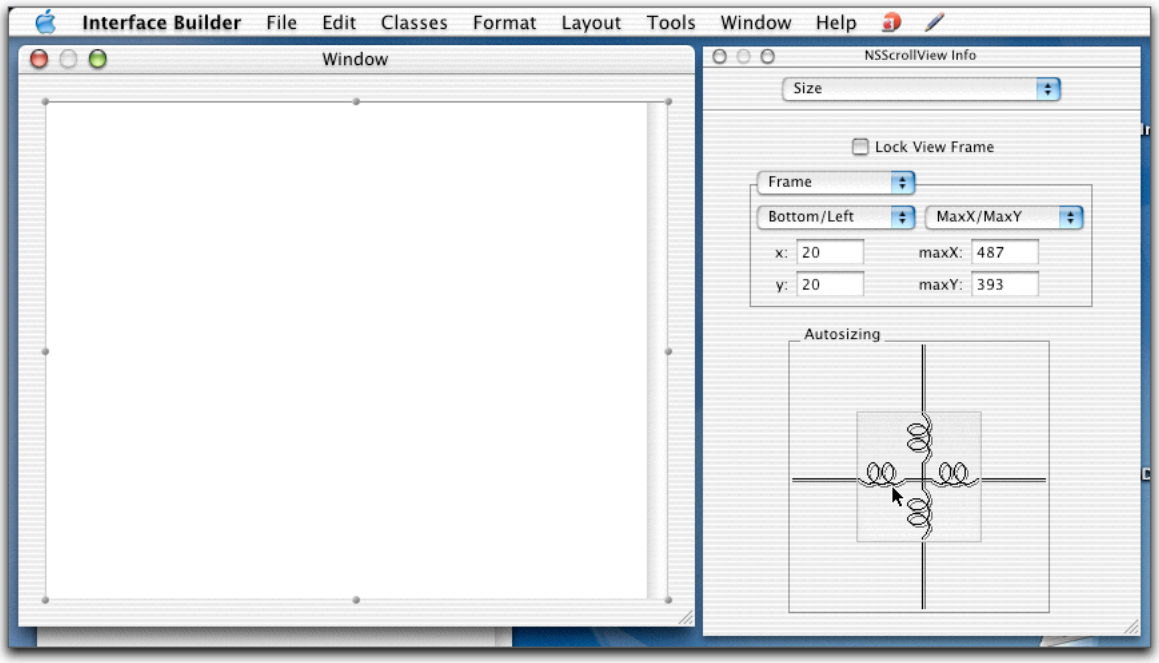
ウインドウも NSTextView も適当な大きさでかまわない。今回は、Cocoa の動作が分かりやすいように、意図的に、NSTextView をウインドウの内側に隙間を持たせて配置した。もちろん、マウスでドラッグなどすることで位置を移動しサイズを変更させることができる。

NSTextView のコンポーネントをウインドウないに広く表示されるように配置した



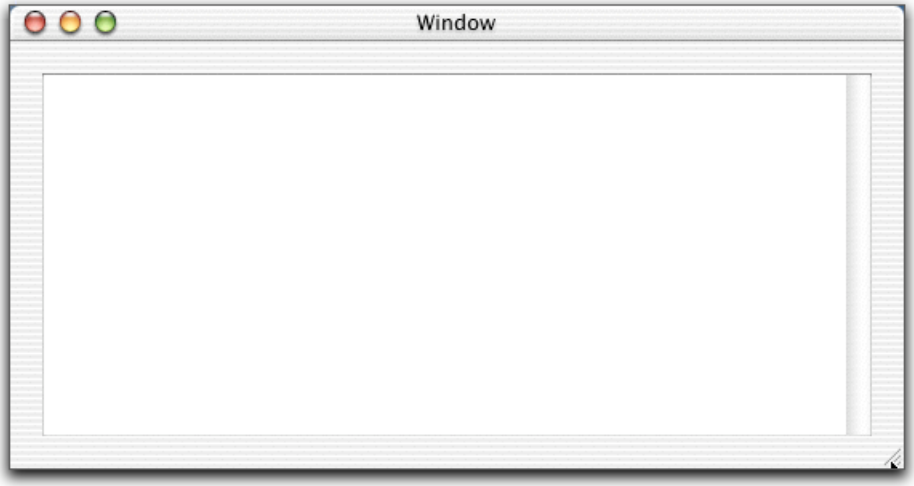
これで実行すれば実はもうキー入力はうけつけてしまうわけだが、このままだと、ウインドウの大きさ合わせて NSTextView の大きさも追随するという動作はできない。NSTextView の大きさはウインドウのサイズに関係なくいつも同じになってしまう。そこで、ウインドウの大きさの変化に追随させたいのであるが、そのためには NSTextView の設定を行うだけでいい。NSTextView が選択されているというか、このコントロールの一番外側の NSScrollView が選択されている状態にする。オブジェクトの周囲のハンドルが四角く、Info パレットのタイトルバーに NSScrollView とでている状態かを確認しよう。あまりクリックすると、その中の NSTextView が選択されてしまうので注意が必要である。そして、Info パレットの Size のページを出して、下側の Autosizing の領域をクリックして設定を変更する。最初はまっすな線になっているが、これは指定した位置に固定ということだ。クリックすると、バネのようなデザインになり、これによって、この図の場合ではコンポーネントのサイズがそれを含むコンポーネント（ここではウインドウ）のサイズに追随するという設定になる。上下左右とも、内側と外側があるが、ウインドウ内のコンポーネントの場合は内側だけバネ状態にしておけばよい。

NSScrollView の Autosizing を有効にする



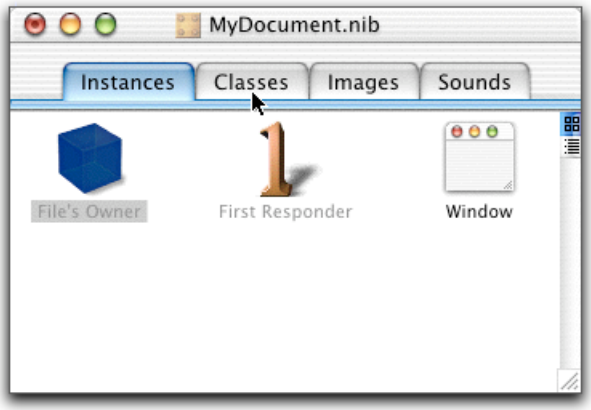
ここで Command+R で Interface Builder 上で実行してみよう。根本的な動作は、コンパイル後でもここでも同じになる。そして、ウインドウが表示され、その中に NSTextView があるのがわかる。ウインドウのサイズを変更すると、それに追随して NSTextView のサイズも変更されていることが分かるはずだ。フレームワークを使うとこうしたところまでも全然プログラミングをしなくてもサポートされている。Toolbox でのプログラミングだと、ウインドウをマウスで操作できるようにするところまでの道のりは長かったわけだが、オブジェクト指向ライブラリはこうした作り込みまでがライブラリ側でできてしまうのである。

すでに NSTextView はウインドウの大きさに追随するように動作している



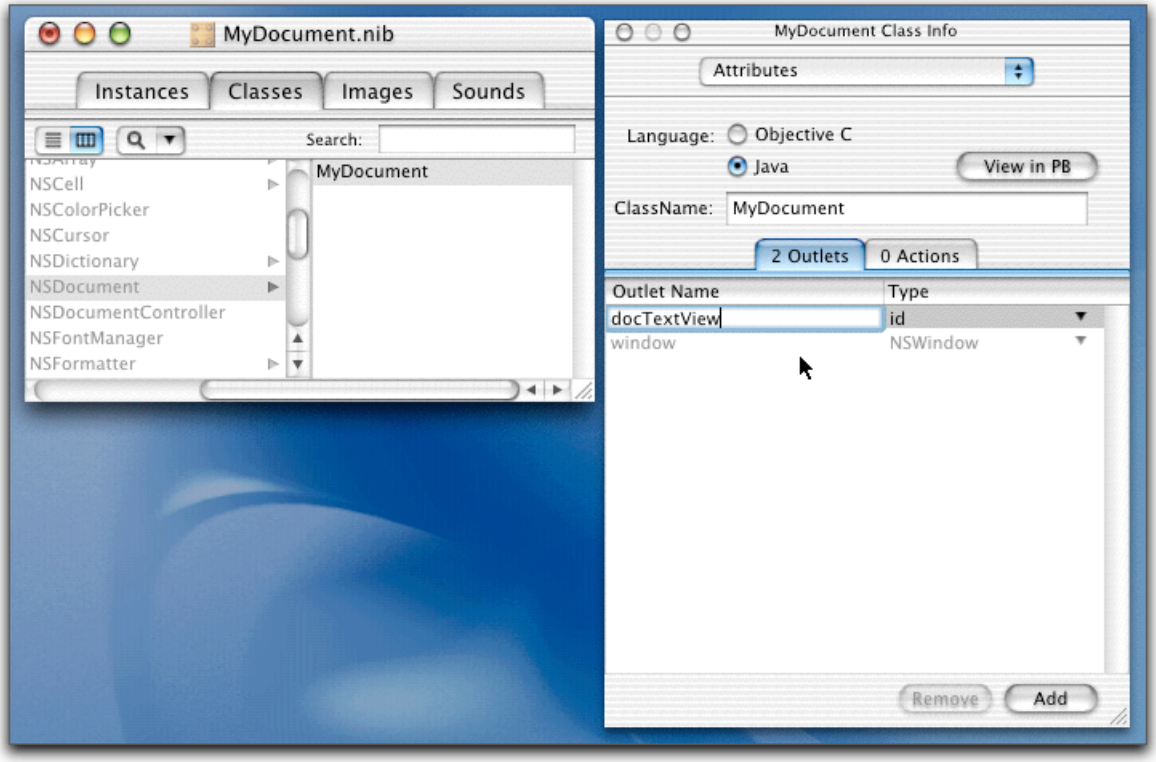
続いて、今配置した NSTextView をプログラムから参照できるようにする。つまり、MyDocument.java 内で NSTextView を操作したいわけだが、そのためには、MyDocument クラスに Outlet を定義すればいい。Instances のタブが選択されている状態で File’s Owner のアイコンを選択してから、Classes のタブをクリックすると、画面が切り替わったときに、MyDocument クラスが選択されているので、該当するクラスをいちいち探さなくてもいいだろう。

MyDocument クラスに Outlet を定義する



Classes のタブでクラス階層が表示され、MyDocument が選択されているところで、Info パレットでは Attributes を選択しておく。そして、Outlets のタブで add ボタンをクリックして Outlet を追加し、その名前を docTextView としておく。ここはキータイプをするが、別の場面で正確に同じ名前をキータイプしないと行けない。そのために、クリップボードにコピーをしておいてもいいだろう。

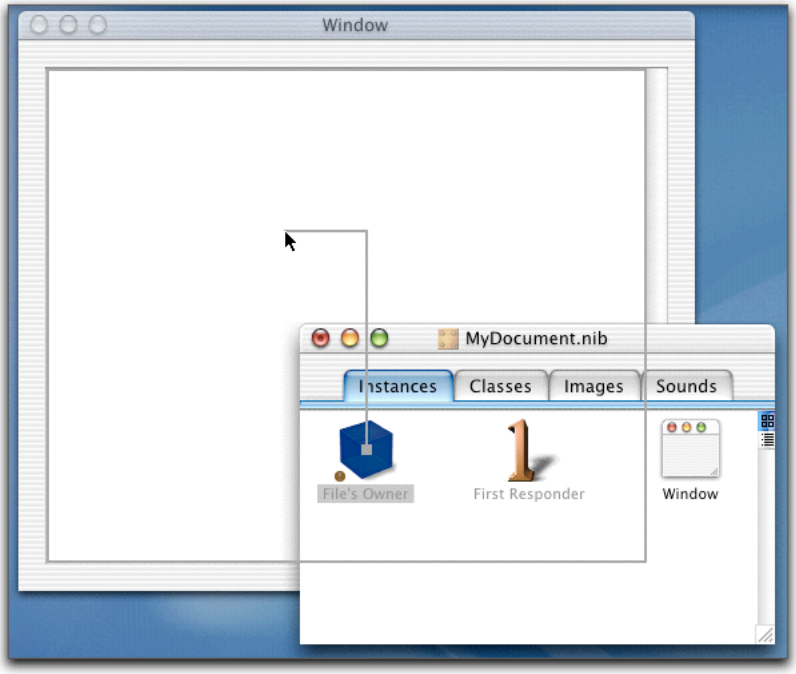
MyDocument クラスに docTextView という Outlet を定義した

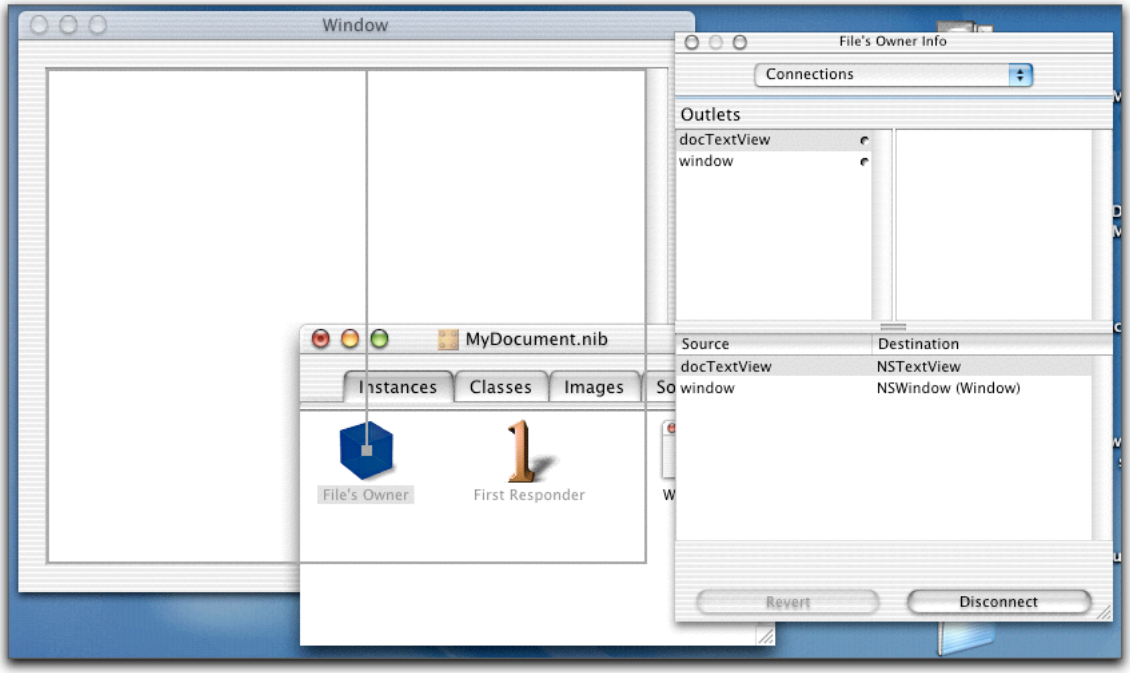


なお、ここで、docTextView の Type についてはとりあえずは id でもかまわないが、ここでは型を選択することもできる。

続いて Outlet の docTextView が、ウインドウに配置した NSTextView を参照できるようにしておく。Instances のタブをクリックしてインスタンスリストを表示し、control キーを押しながら File’s Owner のアイコンから NSTextView のコンポーネントに対してドラッグする。Info パレットでは Connections の情報が表示されるはずだから、Outlets の一覧で docTextView が選択されているのを確認して Connect ボタンをクリックする。そうすれば、docTextView が NSTextView に対して接続されたという設定が、Info パレットの下側に加わる。

Outlet の docTextView が NSTextView を参照するようにリンクする



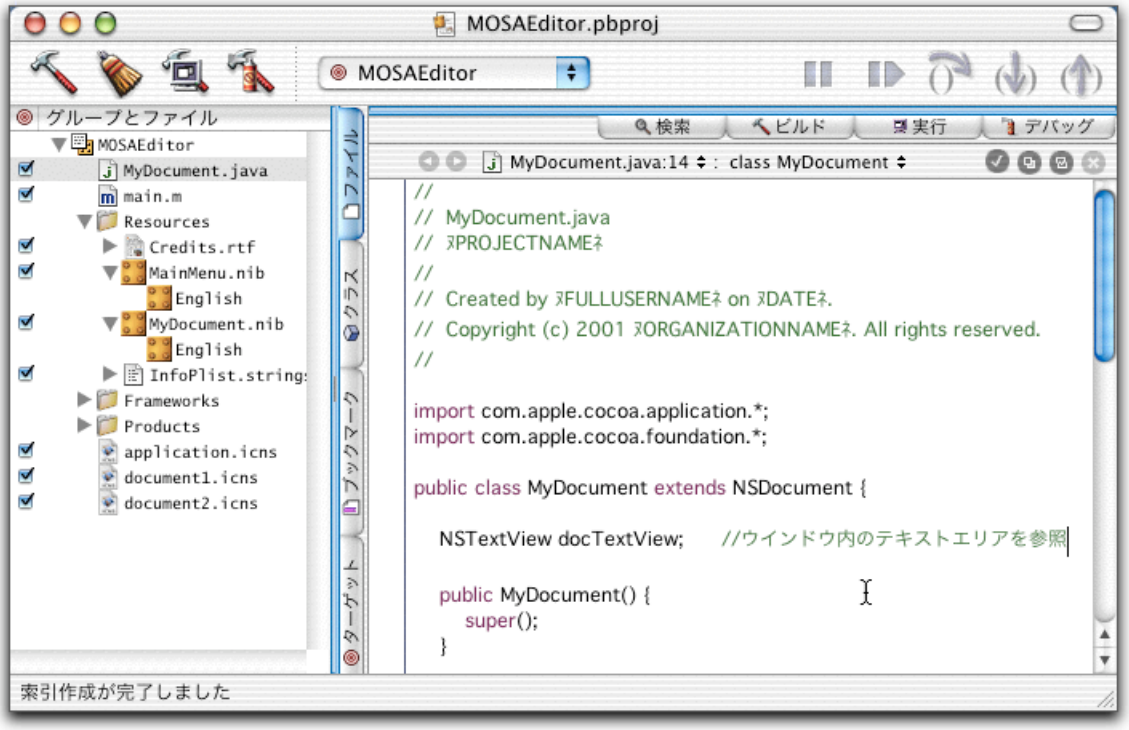


とりあえず、以上で nib ファイルの編集は一段落とする。要は単に NSTextView を配置しただけだが、ポイントとなる設定も先に確認しておいたということである。

Project Builder に戻って、先ほど定義した Outlet に対応するメンバー変数を定義しておこう。MyDocument.java ソースを開き、MyDocument クラスの定義で、メンバー変数として、Outlet と同じ名前の変数を定義する。Java ソースを自動生成させたときは Object 型となるけど、こうしてすでにソースがある場合には自動生成はどうせできないので

あるから、Object 型として定義する必要もない。参照している先のそのものずばり、NSTextView 型で docTextView 変数を定義すればいいのである。

MyDocument クラスに NSTextView の docTextView を定義した



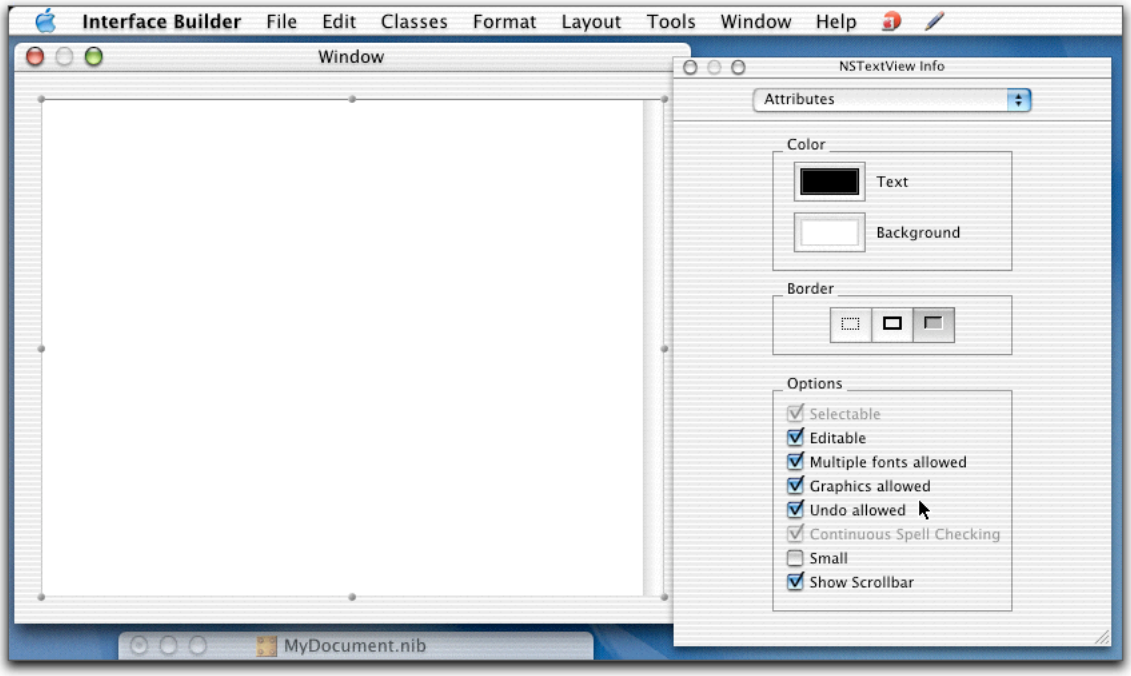
これで、MyDocument.nib ファイルがロードされた段階で、docTextView 変数を通じてウインドウの中の NSTextView を参照できるというわけである。

これでコンパイルして実行すれば、すでにテキストエディタができている。もちろん、保存や開くことができないのではあるが、まだプログラムは 1 行しか書いていないところがミソである。次回からは NSTextView をサマライズしてから、ファイルへの読み書きができるようにしてみよう。

(8)新規にウインドウを開くときの処理

前回の説明で忘れ物があった。文書ファイルに対応した nib ファイルの MyDocument.nib ファイルに、Window というインスタンスがあり、そこに NSTextView を配置したが、その属性を一部変更しておいてもらいたい。

NSTextView の Attributes を変更する

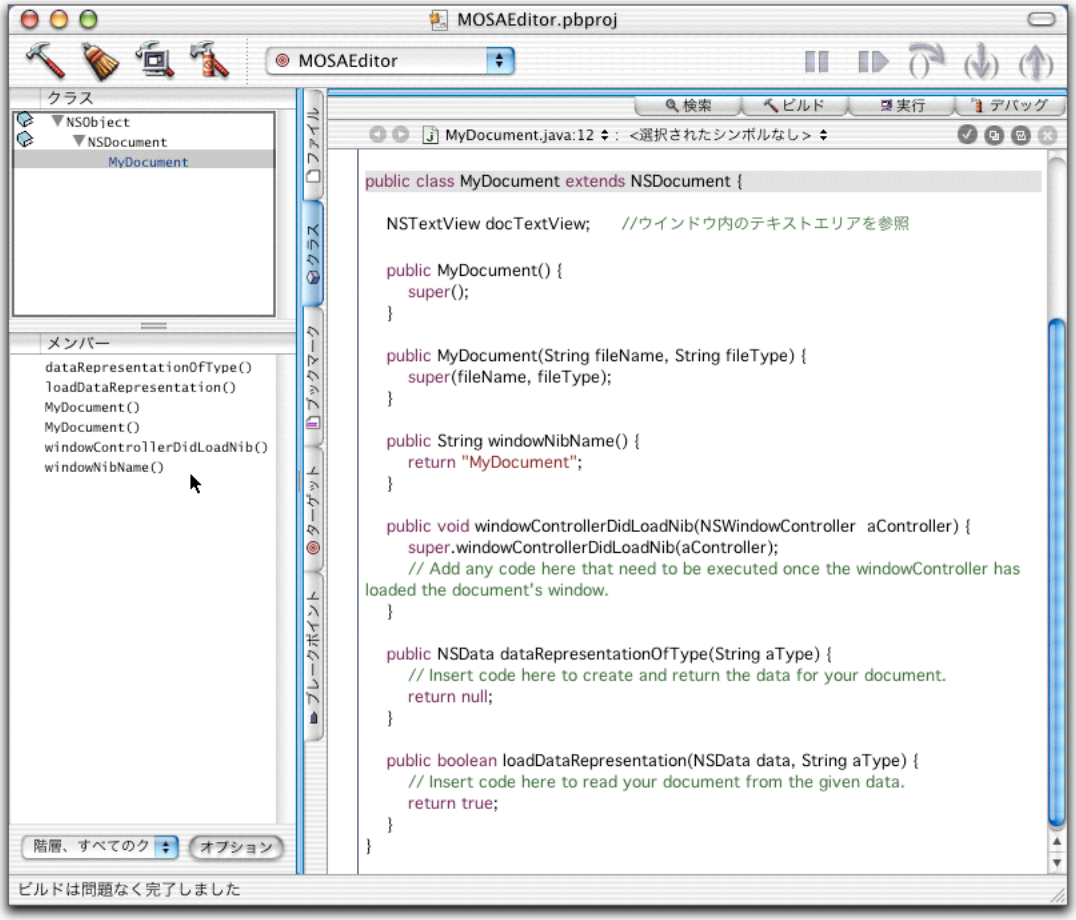


Windows の中にある NSTextView を選択すると、Info パレットに NSTextView の設定が表示される。Attributes をポップアップメニューから選択し、設定を行う。ここで Graphics allowed と Undo allowed のボックスがオフになっているがそれを入れておく。いずれにしても、テキスト編集領域の設定をここで行えるが、属性についてはもちろん、プログラムで変更ができる。

Editable はもちろん、編集作業が可能かどうかで、たとえばテキストの表示だけを行うのなら、このチェックははずせばいいだろう。もし、このチェックをはずせば Selectable が選択できるようになり、編集はできないが選択してコピーなどの作業ができるようになる。Multiple fonts allowed は文字単位でフォント設定を変化させるような書式設定を可能とするかどうかだ。Graphics allowed は、インライングラフィックスの挿入を可能するかどうかを示す。Undo allowed は Undo が可能かどうかだが、実は Revert メニューが使えるかどうもここのチェックに関係しているようである。

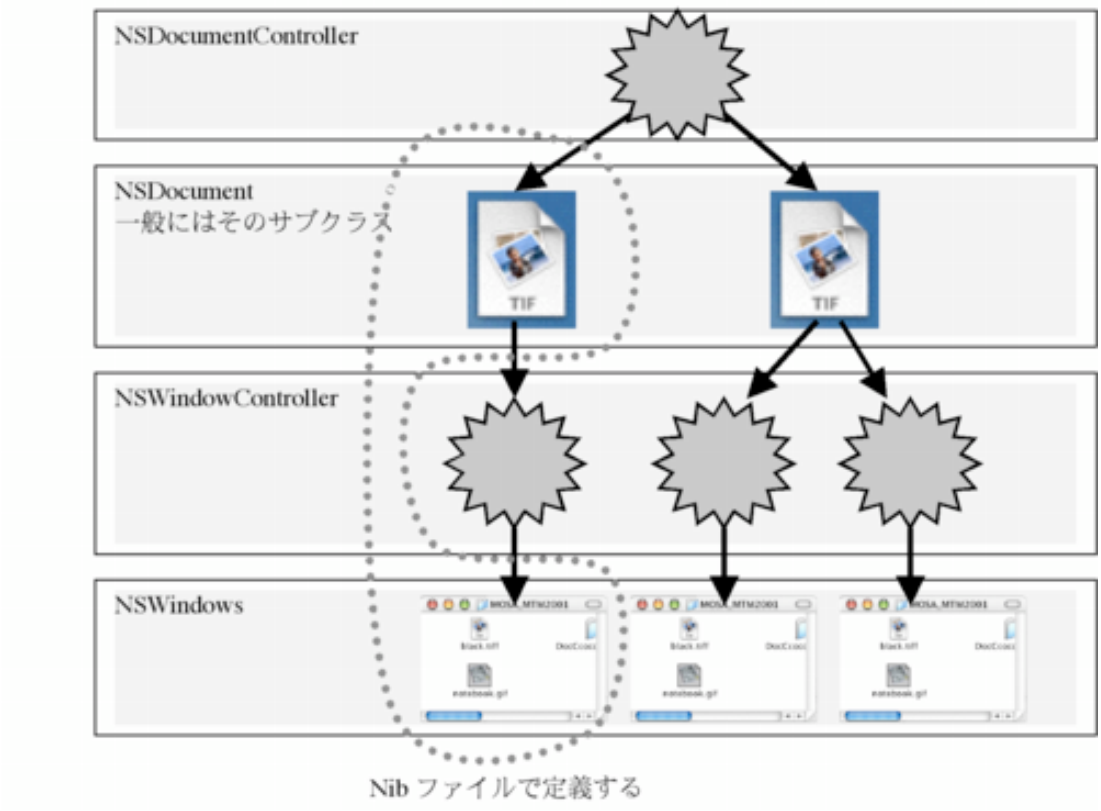
続いてプログラムに移りたいが、今のところはまだ、1 行だけしか追加していない MyDocument.java を見て見よう。すでにいくつかのメソッドが定義されている。

MyDocument.java の最初の状態



ここで改めて、ドキュメントベースのアプリケーションで稼働しているオブジェクト階層を見ながら解説をしよう。以前にも示したが、以下の図だ。

ドキュメントベースのアプリケーション



ここで、ドキュメントを管理する NSDocment のサブクラスとして、MyDocoment クラスが定義されている。NSDocument のサブクラスは必ず作らないといけないと説明したが、そのサブクラスで必ずオーバーライドしなければないメソッドがある。それぞれ、各メソッドで組み込まないといけない機能は以下の通りだ。

（Override-NSDocument） **ウインドウに表示されている文書データを、ファイルに保存できる形式に整える**		
NSData dataRepresentationOfType(String aType);		
戻り値	文書データを NSData 型にしたもの。null を戻すと保存のキャンセル	
引数	aType	保存する文書の型を示す文字列（書類タイプの「名前」）

（Override-NSDocument） **文書ファイルから読み取ったデータをどこかに保存しておく**		
boolean loadDataRepresentation(NSData docData, String docType);		
戻り値	読み込み動作をキャンセルするな false、続けるなら true を戻す	
引数	docData	ファイルから読み取ったデータが得られる
	aType	ファイルの文書の型を示す文字列（書類タイプの「名前」）

（Override-NSDocument） **このクラスに関連付けられている nib ファイルの名前を戻す**
String windowNibName()
戻り値 nib ファイルのファイル名（拡張子は不要）

◇NSDocument（com.apple.cocoa.application）
http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/ApplicationKit/Java/Classes/NSDocument.html

これらのメソッドを定義しないといけないのであるが、すでにテンプレートの状態でメソッドの外枠は作られている。（NSData についてはこの記事の末尾に解説する。）後は中身を書くだけということになっている。

それぞれのプログラムを示す前に、すでに何もプログラミングをしなくても動いている部分を見て見たい。アプリケーションを起動したり、あるいは File メニューから New を選択すると、新たに文書ウインドウが作られる。まだ、文書ファイルとの対応は取られていないが、とにかく設計したユーザインタフェースのウインドウがでてくるのである。ここでは、次のような流れがフレームワーク内で発生している。

1. 起動やメニュー操作によって新しいドキュメントを作るというイベントが発生する。
2. First Responder によって受け取られる。結果的にそれが NSDocumentController で受け取られて処理が行われる。
3. アプリケーション情報をもとに一連のドキュメント対応オブジェクトを構築する。まず、「書類のタイプ」の設定の最初の項目から、文書ファイルは MyDocument クラスで管理することが分かる。（複数ある場合は最初の項目を取り出す）
4. MyDocument の引数のないコンストラクタが呼び出されて、MyDocument オブジェクトが生成される。
5. MyDocument の windowNibName メソッドが呼び出される。これによって、NSDocumentController がロードすべき nib ファイルを知ることができる。そして、ここでは windowNibName で得られた名前をもとに、MyDocument.nib ファイルがロードされる。

6. MyDocument.nib の定義に従ってウインドウが表示され、そこに定義したユーザインタフェースなどが定義通りに画面に表示される。
7. nib ファイルをロード後に MyDocument の windowControllerDidLoadNib メソッドが呼び出される。

最初での First Responder については別途説明するが、簡単に言えば、さまざまなイベントの流れを一元的に受け付けるオブジェクトだと考えれば良い。そして、フレームワークの中で、そのイベントを受け付ける流れができている。たとえば、テキスト編集中はテキストフィールドが受け付けるがそこで処理されないイベントはアプリケーションなどに引き継がれるといった連鎖が自動的に内部で構築されているのである。テンプレートからプロジェクトを作成すると、すでにいろいろなところで「MyDocument」という設定がなされているが、上記の流れをもとに、どのデータをもとにしてどのデータが取り出されるかという関連付けをチェックしておいてもらいたい。複数のドキュメントクラスや nib ファイルを扱う場合にはそうした知識がないといけないだろう。

NSData について

NSDocument でのオーバーライドしなければならなかったメソッドでは、データを NSData 型で取り扱わないといけない。これは、Cocoa の Foundation に定義されたクラスであるが、要はバイト型データの配列を保持してパッケージとして扱うためのクラスである。以下は、コンストラクタと利用するメソッドをまとめておいた。いずれにしても、データは byte 型の配列で得られるので、byte 配列のラッパークラスということである。

NSData クラスのインスタンスを生成するコンストラクタ		
NSData() NSData(byte[] bytes, int start, int length) NSData(byte[] bytes) NSData(java.io.File aFile) NSData(java.net.URL anURL) NSData(NSData aData) NSData(String aString)		
戻り値	生成した NSData オブジェクトへの参照	
引数	bytes	データが入っている byte 型配列
	start	取り出す最初の位置（ないものは byte 配列全部の NSData を作る）
	length	取り出すバイト数（ないものは byte 配列全部の NSData を作る）
	aFile、anURL	ファイルや URL から取り出したデータをもとに NSData を作成
	aString	文字列をもとにした NSData を作成

（以下、《》はそのクラスのインスタンスへの参照を指定することを意味する。）

NSData に含まれているデータの長さを求める	
int 《NSData》.length();	
戻り値	含まれているデータのバイト数

NSData に含まれているデータを取り出す		
byte[] 《NSData》.bytes(int start, int length);		
戻り値	含まれているデータ	
引数	Start	取り出す最初の位置
	Length	取り出すバイト数

◇NSData（com.apple.cocoa.foundation）

http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/Java/Classes/NSData.html

(9)NSTextView の使い方

実際にプログラムを行う前に、今回のメインのコンポーネントである NSTextView の使い方を調べておこう。基本的なドキュメントは、以下のページにある。

◇NSTextView

http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/ApplicationKit/Java/Classes/NSTextView.html

このドキュメントの最初にあるように、NSTextView はいくつかのクラス階層を経て定義されている。階層的には以下のようになっている。ここでの実際のテキスト処理は、NSTextView よりもその基底クラスである NSText という抽象クラスで定義されたメソッドを使うことが多くなる。

NSTextView←NSText←NSView←NSResponder←NSObject

まず、NSTextView からのテキストの取り出しや設定を行うには、以下のようなメソッドが用意されている。単純にテキストのやりとりだけならメソッドもシンプルだ。

（以下、《》はそのクラスのインスタンスへの参照を指定することを意味する。）

編集領域にあるテキストデータだけを取り出す
String 《NSText》.string();
戻り値 NSTextView に表示されているテキスト

編集領域にテキストを設定する		
void 《NSText》.setString(String aString);		
引数	aString	設定する文字列

一方、スタイル付きのテキストやあるいはグラフィックスを埋め込んだテキストについては、次のように、Ritch Text 形式でのデータの出し入れができるようになっている。こちらは汎用的な処理ができるように、NSTextView の中の部分文字列に対する処理

となっている。

NSTextView で扱える Ritch Text は 2 種類ある。通常の Ritch Text（便宜上、「通常」とする）の方と、RTFD という形式だ。RTFD は通常の Ritch Text に加えて画像ファイルの中身もそのままパッケージしたデータである。実際にそれぞれで作業をしてみると、通常の Ritch Text の方はペーストしたグラフィックスが保存されないのに対して、RTFD は保存がされる。ただ、通常の Ritch Text だと Word で読めたのだが、RTFD は読み込めなかった。今回はいずれにしても独自の拡張子を付けているが、NeXT 時代からの作法に従うとそれぞれ、.rtf、.rtfd という拡張子のファイルに保存するようである。

編集領域から Ritch Text でデータを取り出す		
NSData 《NSTextView》.RTFFromRange(NSRange aRange);		
戻り値	取り出した Ritch Text データの NSData 型データ	
引数	aRange	テキストを取り出す範囲

編集領域から RTFD 形式でデータを取り出す		
NSData 《NSTextView》.RTFDFromRange(NSRange aRange);		
戻り値	取り出した RTFD データの NSData 型データ	
引数	aRange	テキストを取り出す範囲

編集領域の一部分に Ritch Text データを設定する		
void《NSTextView》.replaceCharactersInRangeWithRTF(NSRange aRange, NSData rtfData);		
引数	aRange	設定する範囲
	rtfData	設定する Ritch Text データを NSData 型で与える

編集領域の一部分に RTFD 形式のデータを設定する		
void 《 NSTextView 》 .replaceCharactersInRangeWithRTFD(NSRange aRange, NSData rtfData);		
戻り値		
引数	aRange	設定する範囲
	rtfData	設定する RTFD データを NSData 型で与える

なお、範囲を示す引数には NSRange というクラスを使う。これは Cocoa の Foundation

で定義されているクラスであるが、以下にコンストラクタを紹介しておこう。要は、どこから何文字ということを記録するオブジェクトである。オブジェクトの生成を行うのがほとんどだろう。

NSRange クラスのインスタンスを生成するコンストラクタ		
NSRange(); NSRange(int location, int length); NSRange(NSRange aRange);		
戻り値	生成した NSRange オブジェクトへの参照	
引数	location	範囲の最初のポイント
	length	範囲の長さ
	aRange	もとになる NSRange オブジェクト

◇NSRange（com.apple.cocoa.foundation）
http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/Java/Classes/NSRange.html

以上の知識をもとに、まずはファイルへの保存を組み込みたい。ファイルへの保存は、Cocoa フレームワークの中では次のような流れで作業を行なう。

1. 起動やメニュー操作によって保存イベントが発生する。
2. First Responder によって受け取られる。結果的にそれが MyDocoument の saveDocument メソッドで受け取られるが、そのメソッドのオーバーライドがされていないのなら、もとになった NSDocument で処理が行われる。
3. NSDocument の saveDocument メソッドでは、必要に応じて、ファイルを保存するときのシートを表示する。文書ファイルの種類を参照して、扱えないタイプのファイルは一覧でグレーになるなどの処理は自動的に組み込まれている。
4. ファイルを指定するかあるいはすでにファイルが確定されているとする。するとファイルへの書き込み処理が始まる。まず最初に、dataRepresentationOfType メソッドが呼び出される。dataRepresentationOfType メソッドは MyDocument クラスでオーバーライドしているので、そこで書かれたメソッドが処理される。
5. dataRepresentationOfType メソッドが終了すると NSData 型データでフレームワークに保存すべきデータが戻されるので、指定したファイルにそのデータをま

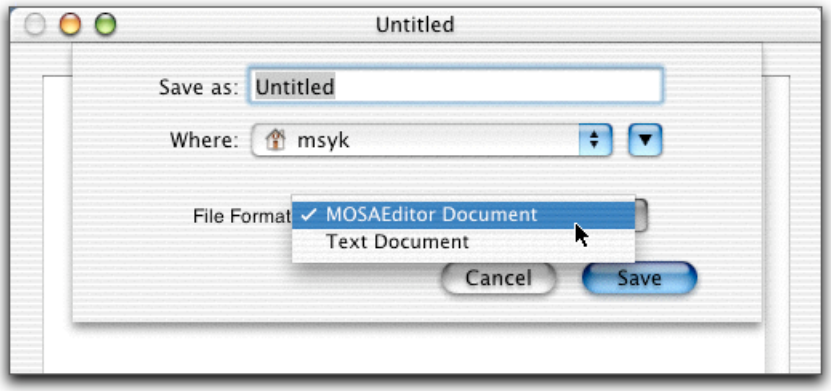
るごと保存する。

ここで、dataRepresentationOfType メソッドを MyDocoment.java の中に次のように定義をした。引数の aType には保存すべきデータの形式が入って呼び出される。

```
public NSData dataRepresentationOfType(String aType) {
    if(aType.compareTo("MOSAEditor Document") == 0)    {
        int textLength = docTextView.string().length();
        NSRange allRange = new NSRange(0, textLength);
        return docTextView.RTFDFromRange(allRange);
    }
    if(aType.compareTo("Text Document") == 0)
        try    {
            return new NSData(docTextView.string().getBytes("x-sjis"));
        }
        catch(Exception e)        {
            System.out.println(e.getMessage());
            return null;
        }
    else
        return null;
}
```

実際にアプリケーションを起動して保存をすると、次の図のように保存するフォルダとファイル名を指定するシートが表示される。ファイルの形式では、Project Builder でのアプリケーション設定で指定した「書類タイプ」の名前が一覧されている。

保存のシートで見られる書類タイプの名前



ここで、どちらかのフォーマットを指定してファイルを保存すると、dataRepresentationOfType が呼びされるが、具体的にはここでは、aType 引数は「MOSAEditor Document」ないしは「Text Document」のいずれかの文字列になっているはずである。そこで、それらの文字列ごとに処理を分岐させているというのが dataRepresentationOfType メソッドの大枠である。

Text Document の場合には、string メソッドで NSTextView からテキストを取り出せばよい。これで入力したテキストが全部取り出される。ただし、それは Unicode のテキストである。ここではシフト JIS コードで保存したいので、getBytes メソッドでコード変換をしつつさらに byte 型配列にしている。そしてその NSData オブジェクトを生成して戻しているということだ。なお、文字コード変換を伴う場合には例外の取得が必要になるが、エラーになったらいちおう null を戻している。

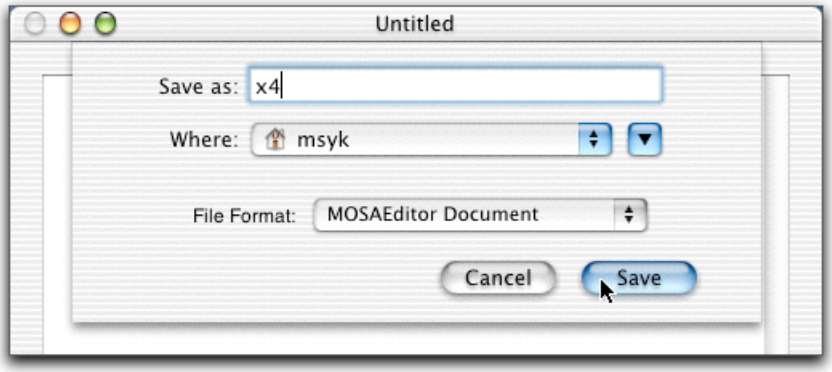
一方、MOSAEditor Document なら、NSTextView の中身を RTFD 形式で取り出したいが、NSRange で範囲を指定しなければならない。「全部」という指定はないから、ここでは NSTextView の長さをチェックしないといけない。そのためには、string メソッドで文字列を取り出し、その長さを length メソッドで得る。これでグラフィクスを含めての範囲が得られるので、それをもとに NSRange オブジェクトを生成する。なお、NSTextView での最初の文字位置は 0 となる。あとは、RTFDFromRange メソッドを使えば、必要なデータが得られるということになる。このメソッドはいきなり NSData 型が得られるので、それ以上することはないということである。

(10)保存結果と文書ファイル

保存ができるようになったところで、保存のときの動作をチェックしよう。しつこいようだが、ここまでのプログラムは、単に「ウインドウに表示されているテキストをNSData 型で戻した」という部分を作っただけである。以下、検討する動作は Cocoa のフレームワークが提供している動作なのである。

まず、Command+S などとにかく保存の作業をすると、最初はシートで保存するフォルダとファイル名を指定する必要がある。別に、何の変哲もないシートではあるが、以下の図は単に「x4」という名前（いい加減な名前で申し訳ない…）を入力している。そして、Save ボタンをクリックしている。

保存するフォルダとファイル名を指定する

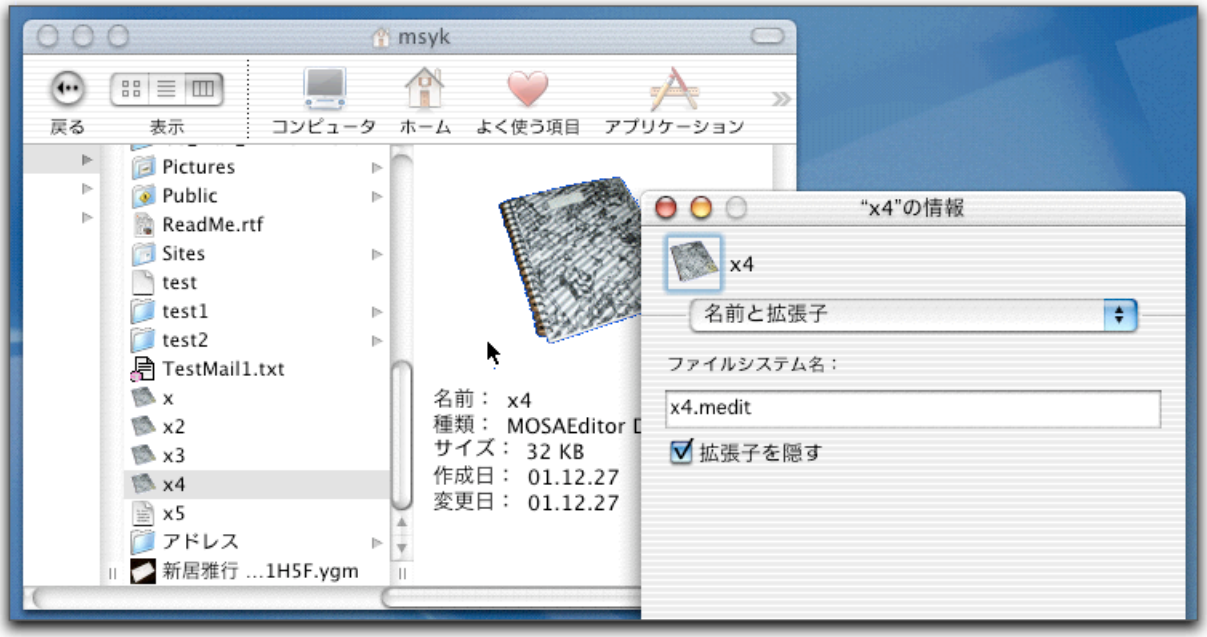


さっそく保存したファイルを Finder でチェックしたいのだが、実はアイコンの反映はすぐには行われない。知り得ている情報では、Mac OS X を起動するときにはハードディスク内をさらってアプリケーションを調べて、そこにある文書情報から、拡張子やファイルタイプとアイコンの対応、つまりデスクトップデータベースを更新しているということである。一方、ログイン後は Applictions フォルダだけをしらべてデスクトップを更新しているということになっている。ところが、プロジェクトのフォルダの中にアプリケーションがある場合再起動でアイコンが反映される場合もあるかもしれないが、変更結果が反映されないこともあるようである。一番確実なのは、アプリケーションを/Applications フォルダにコピーして、ログインをしなおすか再起動することのようである。

というわけで、再起動してから、今保存した x4 というファイルを見ると、Finder でも同じ名前である x4 が見えている。しかしながら、Finder 情報を見ると、ファイル名は、

文書タイプで指定した拡張子が付けられた x4.medit となっている。もちろん、「拡張子を隠す」の属性にチェックが入っている。

保存したファイルを Finder で見てみた

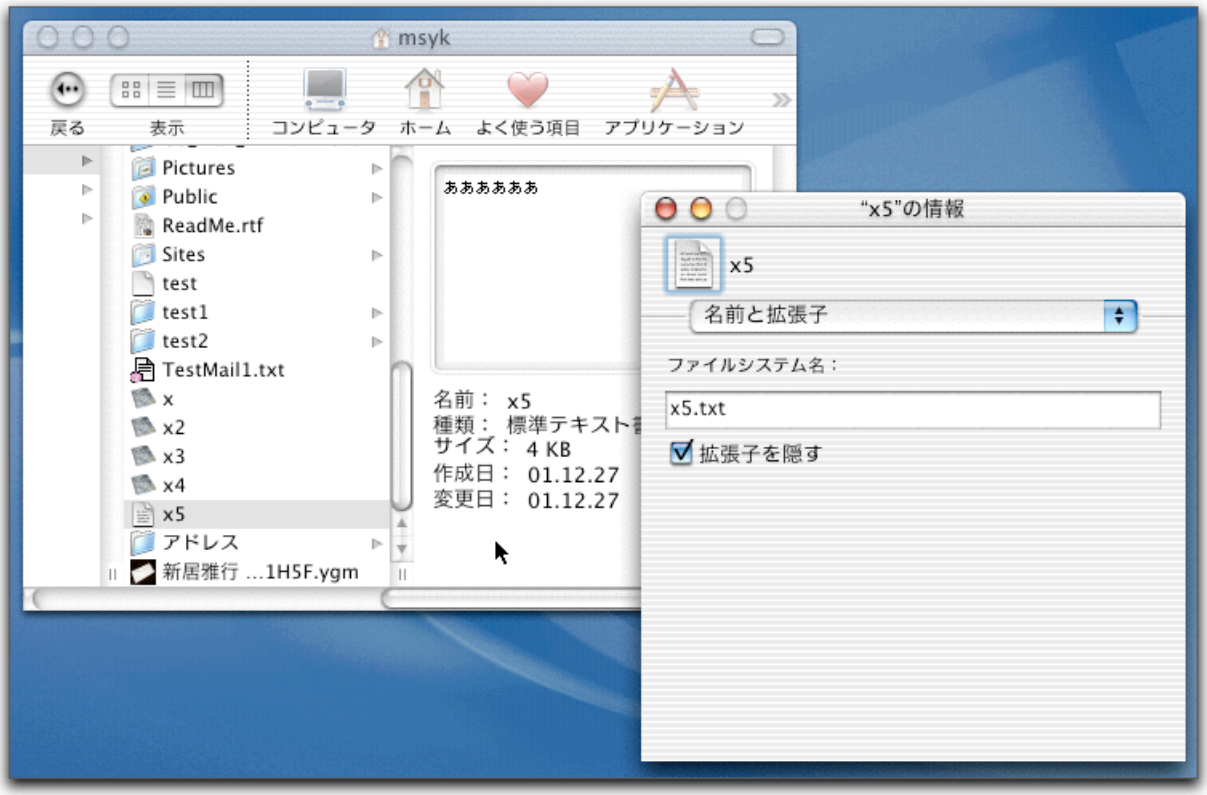


いずれにしても、アイコンは指定したグラフィックスが表示されているのでいいとしよう。ファイル名には自動的に規定の拡張子が付けられ、拡張子は非表示の状態になる。とにかくこれが基本的な動作である。

また、Finder でみる文書の種類は、「MOSAEditor Document」となっていて、つまりは文書タイプの「名前」（もちろん、File Format のポップアップで見られる文字列）がそのまま確認できるようになっている。

同じように、今度は、File Format で Text Document を選択すると、これもやはり文書タイプでの設定とおり.txt の拡張子が自動的に付けられる。Finder 上では拡張子は隠される。ただ、.txt ファイルは、TextEdit の文書となってしまって割り当てたアイコンにはならない。おそらく、システム側での.txt ファイルに対する定義があってそれはどうやっても書き変わらない気がするのである。

テキストで保存したファイルを Finder で見てみた



適当なツールで、ファイルタイプやクリエイタが設定されているかを確認してもらいたいが、この状態では文書ファイルにはまったくクリエイタやファイルタイプは設定されていない。独自のファイル形式で独自の拡張子を使う限りはそれも大きな問題にならないかもしれない。ファイルタイプが設定されていないと Mac OS でのファイルの利用に問題があるとは言え、Cocoa のアプリケーションはいずれにしても Mac OS では稼働しない。それに、ファイルタイプを設定しなくてもアイコンは表示される。ただ、Carbon のアプリケーションでも読み込めるようにしたい場合に、そのアプリケーションがファイルの種類をファイルタイプだけで得ているような場合には問題があるだろう。その場合はいずれにしても、別途プログラムを追加して、ファイルタイプやクリエイタを自分で設定しないといけなくなる。（この方法については別途説明したいが、いい方法をいま探し中である。）

ちなみに、ここでのテキストファイルで独自にアイコンを設定したいとかいった場合には、ファイルタイプとクリエイタを設定することでおそらく可能だと思われる。今のままだと MOSAEditor で作成したテキストファイルをダブルクリックすると、TextEdit で開いてしまうが、ファイルタイプなどを設定すればおそらくは MOSAEditor で開くことができるようになると思われる。

ここでファイルを保存するとき、たとえば「x5.medit」のように、規定の拡張子を付けた場合もチェックしてもらいたい。このとき、Finder では x5.medit のように拡張子も含めたファイル名が表示される。また、「拡張子を隠す」のチェックには、チェックが入らない。つまり、ユーザが明示的にキータイプした場合には、拡張子は表示するという規則になっているわけだ。
さらに、保存時に「x6.mydoc」のように、アプリケーションで登録されていない拡張子を付けた場合は、実際のファイル名は「x6.mydoc.medit」のように、規定の拡張子を自動的に付ける。そして拡張子は隠す設定となり、Finder 上では「x6.mydoc」となるのである。
ファイル名を指定して保存した後、文書のウインドウのタイトルバーを見てもらいたい。Finder で表示されているアイコンで文書が表示され、Finder で表示されている名前がタイトルバーに表示されている。だから、拡張子が非表示なら、タイトルバーにも拡張子は表示されないのである。

ところで、2001/12/8 に開催された MOSA の Macintosh Software Meeting で、Apple からのセッションにおいてこうした拡張子の話が説明されたが、そこでの質問で、拡張子に関する動作のお手本はないのかということが出された。そこでは「ない」と答えられたものの、ある意味では、Cocoa の Document-based アプリケーションは、独自の拡張子を付ける上ではまさに手本であると言えないだろうか。ただ、問題は.txt や.jpg などの汎用的な拡張子を付けたときにどういった動作を意図するかといったところだろう。これについては、ガイドライン的なものは確かに見られないが、逆にガイドライン化しにくいのかもしれない。結果的にはアプリケーションを作る人次第ということなのではないだろうか。

(11)ファイルを開く

続いてファイルを開くようにプログラムを追加しよう。すでに、MOSAEditor 文書が作成されているが、MOSAEditor を終了してから、その文書をダブルクリックしてもらいたい。すると、MOSAEditor が起動して、その文書ファイルを開こうとするはずだ。開いてももちろんファイルの中身を文書ウインドウにセットすることはできないので、ファイルの中身は表示されないが、すでに必要な機能のいくつかは組み込まれている。Finder でのダブルクリックやあるいは Dock 等でのドラッグ&ドロップにより、アプリケーションには OpenDocument の AppleEvent がやってくる。その AppleEvent に対応して、その文書を開くという機能がすでに Cocoa のフレームワークでは組み込まれているわけだ。したがって、AppleEvent に対応するというプログラムは、この機能をそのまま使う範囲ではまったく自分で書く必要はないのである。あとは、アプリケーションのユーザインタフェースに合わせてウインドウにデータをセットする部分だけを記述すればいいわけだ。

まずは、ファイルを開くときの動作の流れを説明しよう。

1. Open のメニューなどを選択することで First Responder に openDocument イベントが伝達すると、NSDocumentController においてそれを取得する。この場合は、開くファイルを指定するダイアログボックスが表示され、ユーザが指定したファイルを開く動作に入る。

- 一方、AppleEvent の OpenDocument イベントや、あるいは Recent Open から過去に開いたファイルの項目を選択すると、やはり NSDocumentController クラスで、指定した文書ファイルを開くという処理に受け継がれる。

2. ファイル情報から、文書タイプと照らし合わせて、その文書を管理するクラスを判読する。つまり、.medit ファイルなら文書タイプの情報から MyDocument クラスであることが分かるので、MyDocument クラスを生成する。このとき、MyDocument(String fileName, String fileType)の方のコンストラクタが呼び出される。
3. ファイルの中身が実際に読み込まれる。Cocoa のフレームワーク側では読み込んだファイルの中身そのままの NSData オブジェクトを作っておく。
4. 生成した MyDocument の loadDataRepresention メソッドを呼び出す。ここでは

読み込んだファイルの中身とファイルの種類が引数で渡される。注意したいのは、ここではまだ nib ファイルはロードされていないということだ。

5. loadDataRepresention メソッドが true を戻すと、windowNibName メソッドが呼び出される。そこで、このクラスで利用する nib ファイル名が知らされる。
6. 指定した nib ファイルをロードしてインスタンス化する。従って、そこに定義したウインドウが実際に表示される。
7. MyDocument の windowControllerDidLoadNib メソッドが呼び出される。ここでは、すでに nib ファイルのロードが終わっている。

簡単に言えば、loadDataRepresention が呼び出され、nib がロードされ、windowControllerDidLoadNib が呼び出されるということだ。loadDataRepresention ではまだ nib がロードされていないので、NSTextView は生成されていない。ここでファイルのデータをセットしたいと思うところだが、それはできないのである。従って、ここではファイルから取り込んだデータをクラス内で覚えておくということになる。そのために、MyDocument クラスのメンバー変数として fileContents、fileType をまず定義しておく。

```
public class MyDocument extends NSDocument {

    public NSTextView docTextView; //ウインドウ内のテキストエリアを参照
    private NSData fileContents;   //読み込んだファイルの中身を記録する
    private String fileType;       //読み込んだファイルの種類を記録する

        :
}
```

次に、loadDataRepresention と windowControllerDidLoadNib のメソッドを以下のようにプログラムした。これらのメソッドは最初から定義されているので、中身だけを書き直すのでいいだろう。これらに加えて、setupWindowFromData というメソッドを作っておく。ここで、ファイルから読み取ったデータを NSTextView に、すなわちユーザインタフェースにセットするプログラムを書いておく。今回のプログラムではそこまでのサブルーチン化は不要だが、後から Revert の機能を組み込むのに必要になるので、

ここで作っておく。

```
public void windowControllerDidLoadNib(NSWindowController    aController) {
        super.windowControllerDidLoadNib(aController);
        setupWindowFromData();
}

public boolean loadDataRepresentation(NSData data, String aType) {
        fileContents = data;
        fileType = aType;
        return true;
}

private void setupWindowFromData()          {
        if(fileContents != null){
            if(fileType.compareTo("MOSAEditor Document") == 0)          {
                NSRange allRange = new NSRange(0,docTextView.string().length());
                docTextView.replaceCharactersInRangeWithRTFD(
                                                        allRange, fileContents);
            }
            else if(fileType.compareTo("Text Document") == 0)
                try {
                   byte contentsBytes[] = fileContents.bytes(0, fileContents.length());
                   docTextView.setString(new String(contentsBytes, "x-sjis"));
                }
                catch(Exception e)    {
                      System.out.println(e.getMessage());
                }
            docTextView.setSelectedRange(new NSRange(0,0));
            fileContents = null;
        }
}
```

NSTextView の編集領域で指定した範囲を選択する		
void 《NSTextView》.setSelectedRange(NSRange aRange)		
引数	aRange	選択範囲を NSRange クラスで指定する

loadDataRepresentation では、単に引数で渡されたデータをメンバー変数に記録しているだけである。windowControllerDidLoadNib では、スーパークラス NSDocument のオーバーライドされた方のメソッドを呼び出す必要があるが、その部分は最初から書き込まれている。あとは、setupWindowFromData を呼んでいるだけである。

setupWindowFromData では実際にファイルから読み取ったデータを NSTextView にセットしている。変数 fileContents は、ファイルの中身を NSData 型で与えられているが、null ならそこにはデータはないということで、何もしないでおいている。これも Revert 対応である。そして、書類の種類ごとに条件分岐しているのは、ファイルへの書き込みと同様だ。Ritch Text は RTFD 形式なので、設定は replaceCharactersInRangeWithRTFD を使うが、1 つ目の引数は書類の範囲である。最初は NSTextView には何も文字が設定されていないので、範囲としては 0 文字目から 0 バイトの範囲を指定すればいいということになるのだが、後からの Revert 対応を考えて、ここでは NSTextView の全ての範囲をファイルの中身とそっくり置き換えるというように汎用的に記述することにしよう。2 つ目の引数は NSData をそのまま使えて便利である。なお、最後に、カーソルの位置を NSTextView の最初の部分に必ず設定されるように、setSelectedRange でのメソッドを加えている。

(12)開いたときにカーソルを点滅

これでいちおうのファイルの読み書きができるようになったが、ファイルを開くとNSTextView はアクティブにはなっていないため、クリックしないとカーソルが中で点滅をしない。これには実はちょっと悩んだ。Visual Basic 的に考えたら、アクティベートするというメソッドがあるのかと思い、クラス階層を一生懸命たどったのであるが、どうも見つけられない。そこで、サンプルコードをみていたら、NSWindow クラスに makeFirstResponder という呼び出しがあるのを見つけた。これは、そのウインドウの First Responder となるコンポーネントを指定するのであるが、文書をよく読むと、マウスダウンイベントを受け付けるレスポンダを置き換えると解説されている。この First Responder はウインドウの中の最初にイベントを受け付けるコンポーネントと言うことであるのだが、つまりは、ユーザの応答を受け付ける意味での“アクティブである”といのは First Responder であるということと同義と考えていいのではないかと思われる。目的としては「NSTextView にカーソルを表示したい」ということであるが、このあたりの事情は知っていないとつかみにくいところだろう。また、ファイルを開いたときには、文書の末尾にカーソルが設定されるので、NSTextView のメソッドを使って選択範囲を指定した。それらの機能を windowControllerDidLoadNib に組み込むと、次のようになる。必要なメソッドの情報も合わせて紹介しておこう。

```
public void windowControllerDidLoadNib(NSWindowController aController) {
    super.windowControllerDidLoadNib(aController);
    setupWindowFromData();
    docTextView.window().makeFirstResponder(docTextView);
}
```

引数に指定したコンポーネントを First Responder にする		
boolean 《NSWindow》.makeFirstResponder(NSResponder aResponder)		
戻り値	First Responder の設定ができれば true、できないと false	
引数	aResponder	First Responder に設定するコンポーネント

コントロールが所属するウインドウを求める
NSWindow 《NSView》.window()
戻り値 コントロールが所属するウインドウへの参照

NSTextView は、NSView、NSResponder をいずれもスーパークラスとして持っているクラスである。なお、makeFirstResponder は、引数に指定したオブジェクトにメッセージを送り、Responder になっていいかどうかの判断や、Responder になったときに行う処理をさせることができる。言い換えれば、そうした処理を独自に書き換えるということも可能だということに他ならない。

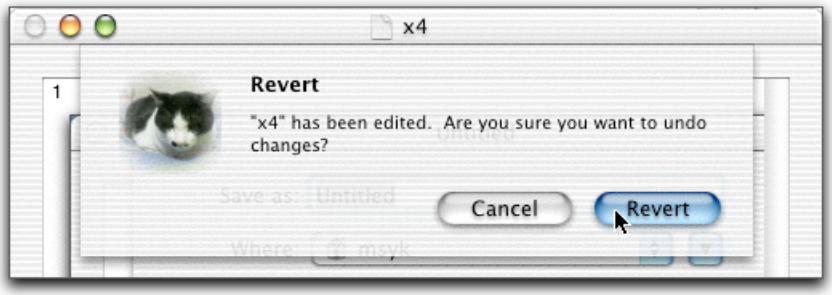
なお、Responder 関連のイベントについては以下の文書で解説されている。

◇Cocoa Programming Topic: Basic Event Handling
http://devworld.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/BasicEventHandling/index.html

(13)Revert を組み込む

続いて、Revert の機能を組み込んでみたい。ちなみに、File メニューから Revert を選択すると、次の図のように、Revert しますかということをシートで問い合わせてくる。

Revert の確認ダイアログボックス



この一連の動作は、NSDocument の revertDocumentToSaved メソッドですでに定義されている。その動作としては次のようになる。

1. Revert メニューを選択するなどして、First Responder に Revert のメッセージが送られる。それが結果的には、MyDocument が受け取るが、そこではメソッドをオーバーライドしていないので NSDocument クラスのメソッドが呼び出される。
2. 変更されているかどうかを確認して、変更されていれば、Revert するかどうかを問い合わせるダイアログボックスを表示する。変更されていなければ、これ以上は何もしない。
3. Revert の動作に入る。すでに文書ファイルなどは確定しているし、nib ファイルもロードされている。ここで、nib ファイルの再ロードは行わないのがポイントだ。
4. MyDocument の loadDataRepresentation メソッドが呼び出される。

nib ファイルはすでにロードされているものを利用する。もちろん、MyDocument はすでにインスタンス化されているから、コンストラクタも呼び出されない。つまり、nib ファイルがロードされた状態で loadDataRepresentation が呼び出されるのである。ファイルの Open だと、nib ファイルがロードされていない状態で loadDataRepresentation が

呼び出されるのと大きく違っている。そこで、こうした動作を両立させるために、まずは、MyDocument クラスのメンバー変数に、nib ファイルがロード済みかどうかをチェックするフラグを設定する。メンバー変数の定義部分は次のようになる。あわせて、windowControllerDidLoadNib メソッドに新たにその変数に値を設定するステートメントを加えた。

```
public class MyDocument extends NSDocument {

    public NSTextView docTextView; //ウインドウ内のテキストエリアを参照
    private NSData fileContents;   //読み込んだファイルの中身を記録する
    private String fileType;       //読み込んだファイルの種類を記録する
    private boolean isNibLoaded = false;   //nib ファイルがロードされたかどうか
        :
    public void windowControllerDidLoadNib(NSWindowController   aController) {
        super.windowControllerDidLoadNib(aController);
        setupWindowFromData();
        docTextView.window().makeFirstResponder(docTextView);
        isNibLoaded = true;
    }
        :
}
```

次に、loadDataRepresentation を以下のように改造する。ここで、変数に応じて、Open なのか Revert なのかを判別できることを利用している。Open の場合には、ここでは NSData などへの参照を記録するだけだが、Revert では実際にデータを NSTextVeiw へ設定する setupWindowFromData を呼び出すということにする。

```
    public boolean loadDataRepresentation(NSData data, String aType) {
        fileContents = data;
        fileType = aType;
        if(isNibLoaded)
            setupWindowFromData();
```

```
        return true;
    }
```

これで、Revert の動作も可能となった。

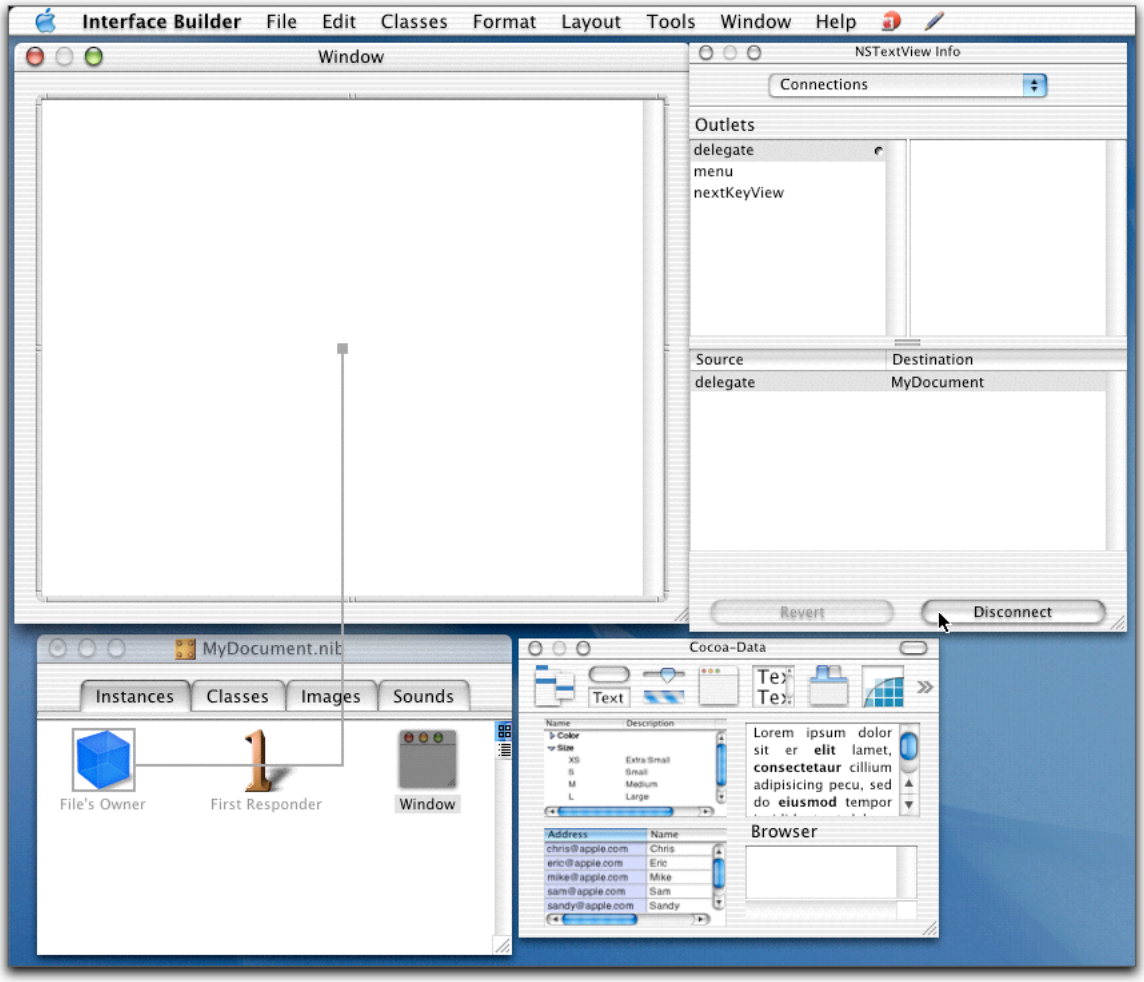
これで概ねは OK なのだが、文書作成アプリケーションとしては次のような不備がある。まず、適当に文書に文字を入れて保存をすると保存はできるが、その後にキータイプをしても、ウインドウの左上にあるクローズの赤ボタンに黒丸が入らない。つまり、Windows に文書が変更されたと言うマーキングが入らない。
さらに、保存もなにもしないで、ウインドウを閉じると、ファイルを保存するかどうかをきちんと聞いてくるのは OK だろう。もう細かくは説明しないが、保存しないのに閉じようとしたときに保存するかどうかを問い合わせるメカニズムも Cocoa には組み込まれている。だが、現状では、一度保存をした後、キータイプをしてそのままウインドウを閉じると、何の警告もなくてウインドウが閉じる。そして、保存後にキータイプした結果は残っていないのである。つまり、ドキュメントに変更があるという情報がどうもセットされていないようなのである。
これらを解決する必要があるのだが、MOSAEditor では NSTextView だけを使っているので、とりあえず、NSTextView が変更されたときに何らかの処理がされるようにしておけば良いと言うことが分かる。そこで、NSTextView そして NSTextView のドキュメントを探ると、textDidChange というデリゲート向けのメソッドがあることが分かる。このメソッドはキータイプなどでの編集があったときに呼び出されるが、setString などでは呼び出されない。おそらく、ユーザの変更作業があった場合に呼び出されるということでよさそうだ。

NSText（NSTextView）で内容が変更されたときに呼び出される（Delegate）		
public void textDidChange(NSNotification aNotification)		
引数	aNotification	変更された NSTextView への参照

そこで、NSTextView でのデリゲートの受付先を設定しておく。新たにクラスを作ってもいいのだが、ここでは簡単のために、MyDocument、つまり nib ファイルでは File's Owner を指定することいしよう。Project Builder から MyDocument.nib ファイルをダブルクリックして開く。Interface Builder で作業をするが、ウインドウの中にある

NSTextView からリンクをしないといけない。そのためには、NSTextView のコンポーネントをダブルクリックして、control キーを押しながら File’s Owner にドラッグする。ここでクリックだけだと、NSTextView を包含している NSScrollView が選択されるのでダブルクリックして、そしてその状態でボックスの内部から control キーを押しながらドラッグする。そして、Outlet で delegate が選択されているのを確認して、Connect ボタンをクリックする。

NSTextView のデリゲート先を指定する



こうして、NSTextView でユーザが編集作業を行うと、MyDocument クラスの textDidChange メソッドが呼び出される。そこで、MyDocument クラスに、以下のようなメソッドを付け加えておく。新たにでてきたメソッドとともに解説しておこう。

```
public void textDidChange(NSNotification aNotification)        {
    docTextView.window().setDocumentEdited(true);    //実は不要
    updateChangeCount(ChangeDone);
```

}

ウインドウに表示された文書が変更されたかどうかをセットする		
void 《NSWindow》.setDocumentEdited(boolean flag)		
引数	flag	true なら変更されたことになる

ドキュメントの変更に関するカウントを更新する		
void 《NSDocument》.updateChangeCount(int changeType)		
引数	changeType	以下の定義定数を指定する ChangeDone：カウンターを増加 ChangeUndone：カウンターを減少 ChangeCleared：カウンターを 0 にする

ここで、NSDocument では、文書を変更したかどうかをカウンターで管理している。変更があれば、カウンターをアップし、Undo によりカウンターをダウンするといった動作を行っている。そして、カウンターが 0 なら修正はされていないという判断ができるという具合だ。ここでは、とりあえずはキータイプすればカウンターがアップすれば変更したことが伝えられるのでそのようにしておく。なお、updateChangeCount の呼び出しを行えば、内部的に setDocumentEdited メソッドも呼び出されているようで、setDocumentEdited は利用しなくてもかまわない。

なお、定義定数はそのまま「ChangeDone」として使える。Java 的には「NSDocument.ChangeDone」なのであろうけども、MyDocument は NSDocument のサブクラスだから、クラスの指定は不要なのである。ちょっと楽ができるみたいな気分（笑）が味わえる。

これで概ね動作上は OK なところまで来たと思われる。保存していないウインドウを複数開いて、Command+Q で終了をしてもらいたい。すると、変更結果を破棄するとか、各ウインドウで変更するかどうかを問い合わせるかを選択するダイアログボックスも、Cocoa のフレームワークで自動的にでてくる。Undo についても、ある程度は動くのであるが、NSTextView の動作上の問題と思われるが、日本語のテキストの場合はきちんと動作しないということも分かる。また、Undo は 1 レベルまでとなっている。多段階の Undo といった高度なことは自分で作成するしかないようだ。

(14)Cocoa でのファイル情報

Cocoa の Document-based Application、すなわち文書ファイルを含むアプリケーションを作る機能を使ったエディタ制作の続きである。正月後に保存したファイルにファイルタイプとクリエイタをつけるというテーマに実は取り組んでいたのである（ちょっとだけはまりながら…）。

Project Builder の「アプリケーション設定」での「書類のタイプ」では、ファイルタイプも記述できたし、ファイルのクリエイタも設定できた。しかしながら、Cocoa のフレームワークでの文書ファイル保存の処理には、ファイルタイプとクリエイタの設定は行われなかったのである。その部分は、プログラマが自分で作らないといけないのである。そのために、文書ファイルを管理するクラスのもとになっている NSDocument クラスの特定のメソッドをオーバーライドしないといけないのであるが、それについては、次回に説明し、ここでは、Cocoa-Java 特有のファイル処理について解説しておきたい。つまり、ファイル処理に必要なメソッドをまずはまとめておきたいのである。

Objective-C でのプログラミングでは、Cocoa に用意されたクラスを使ってファイルの処理を行う。一方 Java の場合は、Java の標準ライブラリである java.io.*あたりに定義されている File や FileInputStream あたりを使ってのファイル処理ができる。それに加えて、Cocoa-Java 特有のファイル処理クラスがあったり、さらには NSData のようにファイル書き込み機能があるなど、いろいろなクラスにファイル関連処理機能がある。いずれにしても、java.io.*は基本として押さえておく必要があるが、これは Java Watch on the X のコーナーでいずれとりあげるつもりである。さらに、Cocoa-Java アプリケーションでは Cocoa 特有のファイル処理を知っておく必要もでてくるので、そのつもりでドキュメントを見ておくべきだろう。

ここで、Java 標準ライブラリでは、File クラスによってファイルを参照するが、Cocoa はパスの文字列を利用してファイルを特定する。パスはもちろん BSD でのパスであるので、Finder で見える階層のものではない。もちろん、File クラスの getPath メソッドでパスは得られるし。パスの文字列を引数にした File クラスのコンストラクタを呼び出すと、パスから File クラスの生成もできるので相互利用は比較的容易である。Cocoa 特有のファイル処理については以下の文書を参照してもらいたいがまだ文書は全部できていないようだ。Objective-C と Java で用意されているクラスがまったく違うところ

に注意をしてもらいたい。Objective-C では NSFileManager というクラスをよく使うのであるが、それに相当する Java のクラスがないのである。

◇Cocoa Programming Topic: Low-level File Management
http://devworld.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/LowLevelFileMgmt/index.html

Cocoa-Java 特有のファイル処理機能はたくさんあるが、ここではファイルタイプとクリエイタに絞って機能を紹介しよう。
実は、最初は MRJ にある FileUtils クラスのクラスメソッドである setFileTypeAndCreator を使えばいいと思ったのであるが、なぜか、

2002-01-05 20:34:32.277 MOSAEditor[6251] *** _NSAutoreleaseNoPool(): Object 0x3658210 of class NSView autoreleased with no pool in place - just leaking

といったメモリのエラーがでてしまう。おそらく Cocoa-Java のクラスをインスタンス化したときには、オートリリースが設定されたメモリブロックとして確保するのだと思われるが、MRJ の利用でそのオートリリースプールが壊されるのだと想像される。ただ、必ずダメというわけでもなく、あるとき作ったプログラムでは Cocoa-Java なのに MRJ のクラスを呼び出して利用できたりもしたことがある。ちょっと謎なのだが、いずれにしても、ファイルタイプやクリエイタをはじめ、ファイルの様々な情報を取り出すための機能が Cocoa-Java に用意されているので、それを利用するのが問題は少ないだろう。

Cocoa でのファイル情報関連機能

NSPathUtilities に次のような機能がある。ちなみに NSPathUtilities クラスは、パスを扱う場合に便利なメソッドがいろいろあるので、Cocoa-Java でのファイル処理では有用に使えることも多いだろう。

指定したファイルにファイル情報を設定する（Static）		
boolean NSPathUtilities.setFileAttributes(String path, NSDictionary attributes);		
戻り値	設定できれば true、失敗すれば false	
引数	path	設定を行うファイルのパス
	attributes	ファイル属性を NSDictionary クラスで指定する

指定したファイルの情報を取得する（Static）		
NSDictionary NSPathUtilities.fileAttributes(String path, boolean flag);		
戻り値	ファイル情報が含まれた NSDictionary クラスのインスタンス（ファイルが存在しない場合、戻り値が null となる）	
引数	path	情報を得たいファイルのパスを指定する
	flag	path で指定したファイルがシンボリックリンクの場合、flag が true ならリンク先のファイルの情報を得る。false なら、シンボリックリンクファイル自体の情報を得る

ここでファイルの属性は、NSDictionay クラスで管理される（このクラスは記事の末尾で説明する）。キーワードと情報についての対応は次の表のようになっている。データのクラスの Integer や Boolean は java.lang パッケージにあるものだ。

属性データの NSDictionary のキーとデータ

setFileAttributes メソッドで設定できる属性
↓　fileAttributes メソッドで取得できる属性
↓　↓　キーワード（データのクラス）データの内容
↓　↓　――――――――――――――――――――――――――――――
×　○　FileSize（Integer）ファイルのサイズのバイト数
○　○　FileModificationDate（NSDate）ファイルの修正日
×　○　FileOwnerAccountName（String）ファイルのオーナー
×　○　FileGroupOwnerAccountName（string）ファイルのグループ
×　○　FileReferenceCount（Integer）ハードリファレンスの個数
×　○　FileIdentifier（Integer）ファイル ID
×　○　FileDeviceIdentifier（Integer）ファイルデバイスの ID
○　○　FilePosixPermissions（Integer）アクセス権
×　○　FileType（String）ファイルの種類（別表参照）
○　○　FileExtensionHidden（Integer）0 なら拡張子を非表示、1 なら表示

○　○　FileHFSCreatorCode（Integer）クリエイタ（未設定なら 0）

○　○　FileHFSTypeCode（Integer）ファイルタイプ（未設定なら 0）

ファイル情報でキー「FileType」に対応するデータとして取り得る文字列

FileTypeDirectory
FileTypeRegular
FileTypeSymbolicLink
FileTypeSocket
FileTypeCharacterSpecial
FileTypeBlockSpecial
FileTypeUnknown

setFileAttributes メソッドでは当然ながら設定可能な情報は限られているけど、さらに、実際に設定できるのは、実行しているプロセスのアカウントと同じオーナーのものに限られる。あるいはルート権限で実行しているプロセスからの変更となる。自分で保存したファイルについては通常はファイルの情報を変更できるはずだ。setFileAttributes メソッドではすべての属性を設定する必要はなく、設定をしたい属性だけ、NSDictionay のキーとして設定すればいい。具体的には次回にプログラムを紹介しよう。
なお、ドキュメントでは整数で得られるデータは integer と記載があるが、実際には、java.lang.Integer という int のラッパークラスとなっている。拡張子を隠しているかどうかは FileExtensionHidden というキーで得られるはずとなっていてドキュメントではクラスについては boolean となっているが実際には java.lang.Integer である。

さて、ファイルタイプやクリエイタは、Integer クラスである（つまりは int 型）。C 言語だと、‘TEXT’と書けば long 型が得られたのでいいのであるが、Java ではこの書き方ができない。そこで、テキストで記述した 4 バイトのコードを整数にしたり、あるいは逆を行う必要がある。そのために用意されたのが、NSHFSFileTypes クラスのメソッドである。

４バイトの文字列から対応する整数値を得る（Static）		
int NSHFSFileTypes.hfsTypeCodeFromFileType(String fileType);		
戻り値	タイプやクリエイタに対応した整数値	
引数	fileType	ファイルタイプやクリエイタを示す文字列（以下の本文を参照）

整数値から４バイトのコードを得る（Static）		
String NSHFSFileTypes.fileTypeForHFSTypeCode(int typeCode);		
戻り値	ファイルタイプやクリエイタを示す文字列（以下の本文を参照）	
引数	typeCode	タイプやクリエイタに対応した整数値

指定したファイルのファイルタイプを取得する（Static）		
String NSHFSFileTypes.hfsTypeOfFile(String filePath);		
戻り値	ファイルタイプを示す文字列（以下の本文を参照、未設定なら ‘’ のみ）	
引数	filePath	調べたいファイルのパスを文字列で指定する

ここで、ファイルタイプを TEXT にしたいのなら、その TEXT に対応した整数値を得るために、4 バイトのコードの前後にシングルクォーテーションを付けた文字列を指定する必要がある。つまり、

```
int theCode = NSHFSFileTypes.hfsTypeCodeFromFileType(" ‘TEXT’ ");
```

とするのである（実際にはシングルクォーテーションは半角）。また、fileTypeForHFSTypeCode や hfsTypeOfFile で得られる文字列も前後にシングルクォーテーションがついているので、結果的に得られる文字列は 6 バイトということになる。

いずれにしても、具体的なプログラムを示さないと、まったく初めてこれらのクラスに遭遇した場合には理解しづらいだろう。NSDictionary クラスのことを説明した後に、実際のプログラムを紹介しよう。

NSDictionary クラス

NSDictionary クラスは、immutable つまりいったん作成すると内容が変更されないこと

が保証されたクラスで、特定のキーに対応するデータを管理することができるクラスだ。java.util.Properties（ないしは HashTable、Dictionary）や、あるいは Perl での連想配列と同じようなものであるが、それの Cocoa 版である。データを複数記憶できる、個数は可変である。それらのデータを取り出すのに、キーというデータを指定する。あるデータにキーが割りあってあってキーを手がかりに取り出すということができる。また、キーやデータをまとめて取り出すという機能がある。コンストラクタは次のように定義されている。4 つあるが、immutable なので、引数無しで空の NSDictionary を作ることは実用上はまずないだろう。4 つ目は事実上の NSDictionary クラスのコピー処理を行うことになる。データを記録させるには、2 つ目ないしは 3 つ目を使うことになる。なお、キーとデータのペアを後から追加することはできないので、コンストラクタで生成時に、データを詰め込んでしまわないといけない。

NSDictionary クラスのインスタンスを生成する（コンストラクタ）		

new NSDictionary();
new NSDictionary(Object[] objects, Object[] keys);
new NSDictionary(Object anObject, Object aKey);
new NSDictionary(NSDictionary otherDictionary);		

戻り値	生成した NSDictionary クラスのインスタンスへの参照	
引数	objects	データ
	key	データに対応するキー
	otherDictionary	これと同じ内容の NSDictionary を生成する

キーとデータはいずれにしても配列で指定する。Object 型だから、つまりは Java のクラスであれば何でもいいということになる。一方、基本型の配列はそのままでは設定できないということになるだろう。たとえば、year というキーに「2002」、month というキーに「1」、day というキーに「8」というデータを割り当てるとすると、例えば次のようなプログラムとなる。整数データは、java.lang.Integer クラスを使えばいいだろう。データとキーをそれぞれ配列にするので、対応がきちんとなるように順序に気をつけて配列を作ればよく。

```
String keys[] = {"year", "month", "day"};
Integer data[] = {new Integer(2002), new Integer(1), new Integer(8)};
NSDictionary dict = new NSDictionary(data, keys);
```

NSDictionary にあるメソッドのうち、よく利用するものを以下にまとめておく。他にもあるが、一般にはここで紹介したものを使うのが一般的だろう。

キーとデータのペアの個数を求める	
int 〈NSDictionary〉.count();	
戻り値	データの個数

キーに対するデータを求める		
Object 〈NSDictionary〉.objectForKey(Object akey);		
戻り値	データ（存在しないキーの場合は null となる）	
引数	key	キー

すべてのキーを Enumeration クラスで得る	
java.util.Enumeration 〈NSDictionary〉.keyEnumerator();	
戻り値	キーのリスト

たとえば、以下のようなプログラムで、変数 path で指定したファイルのファイル情報を、標準出力にキーとデータのペアとして書き出す。

```
import java.util.*;
        :
String path = "/Users/msyk/text.txt";
NSDictionary dic = NSPathUtilities.fileAttributes( path, false);
Enumeration enum = dic.keyEnumerator();

while (enum.hasMoreElements()) {
    Object key = enum.nextElement();
    Object obj = dic.objectForKey(key);
    System.out.println("key="+key.toString()+", data="+obj.toString());
}
```

(15)ファイルタイプとクリエイタを設定する

Cocoa-Java でのファイル処理クラスを紹介したが、それを受けて、Document-based アプリケーションで、ファイルタイプやクリエイタを設定する方法を説明したい。文書ファイルを保存するという機能は NSDocument クラスに組み込まれているが、少々複雑であるもの、ファイル書き込み処理はカスタマイズできるようになっている。NSDocument では、ある文書を保存するときに、指定したファイルにいきなり書き込むのではなく、一時的に別のフォルダにファイルを書き込んで、その後に、書き込んだファイルをユーザが指定したフォルダ位置にあるファイルとして移動するという作業を行なう。したがって、一時的に書き込むファイルをそのまま適当に残しておけば、バックアップファイルを作成すると言う機能までも組み込めるのである。
こうした仕組みは、NDDocument では次のようなメソッドで処理される。ファイルの書き込みを行うときには、writeWithBackupToFile というメソッドが実行され、そこから writeToFile の 4 つの引数があるメソッドが呼び出され、さらにそこから writeToFile の 2 つの引数のあるメソッドが呼び出される。その後に、以前に説明した dataRepresentationOfType メソッドが呼び出されるのである。以下は、NSDocument に組み込まれている機能としてまとめておく。

文書の内容を指定したファイルに保存する		
boolean 〈NSDocument〉.writeWithBackupToFile(String fullDocumentPath, String documentTypeName, int saveOperationType);		
戻り値	書き込みが成功したら true、失敗したら false	
引数	fullDocumentPath	保存する書類ファイルのフルパス
	documentTypeName	ドキュメントの種類を示す文字列
	saveOperationType	保存か名前を付けて保存かなどを示す

文書ファイルのバックアップを残すかどうかを判定する	
boolean 〈NSDocument〉.keepBackupFile();	
戻り値	false を戻す

writeWithBackupToFile は、もし、keepBackupFile メソッドの戻り値が false なら、バックアップファイルは残さない。NSDocument に定義された keepBackupFile は false を戻

すのであるが、これを true を戻してバックアップファイルを残すようにしたいのなら、NSDocument を継承したクラス（ここでは MyDocument クラス）で、keepBackupFile メソッドをオーバーライドして、単に true をモドすようにすればいい。

文書をファイルに書き込む		
boolean 〈 NSDocument 〉 .writeToFile(String fullDocumentPath, String documentTypeName, String fullOriginalDocumentPath, int saveOperationType);		
戻り値	書き込みが成功したら true、失敗したら false	
引数	fullDocumentPath	実際に書き込みを行う文書ファイルのパス名
	documentTypeName	文書の種類を示す文字列
	fullOriginalDocumentPath	ユーザが指定した文書ファイルのパス
	saveOperationType	保存か名前を付けて保存かなどを示す

文書をファイルに書き込む		
boolean 〈NSDocument〉.writeToFile(String fileName, String type);		
戻り値	書き込みが成功したら true、失敗したら false	
引数	fileName	実際に書き込みを行う文書ファイルのパス名
	type	文書の種類を示す文字列

要は、次のようなメソッドの連鎖が行われるのである。

```
writeWithBackupToFile
└→writeToFile(4)
    └→writeToFile(2)
        └→dataRepresentationOfType
        ┌──┘
    ┌──┘
┌──┘
└→keepBackupFile
┌──┘
```

これらのうち、dataRepresentationOfType は必ずオーバーライドして定義しないといけないメソッドだが、あとは必要に応じてオーバーライドする。その意味ではデフォルトの動作を知っておく必要はあるだろう。ここで、ユーザは実施に保存したいファイ

ルのファイル名とフォルダを指定するのだが、それは、writeToFile での引数 fullOriginalDocumentPath としては得られる。一方、writeToFile での引数 fullDocumentPath あるいは fileName では、実際に保存を行う一時的なファイルの名前が得られるのである。たとえば、保存するファイルが、

```
/Users/msyk/b.medit
```

だったとした場合、たとえば、fileName 引数で得られる実際に書き込みを行うファイルのパスは、

```
/private/tmp/501/Temporary Items/com.apple.NSDocument_6567_32252318_1/b.medit
```

といったものだ。/tmp ディレクトリにあることや、ファイル名自体は実際のファイル名と同じであるあたりがポイントになるだろう。
dataRepresentationOfType ではファイルに保存すべきデータは NSData 型で作っておくが、writeToFile では、その NSData 型データを、ファイルに保存するというわけである。
もし、NSDocument で想定されているような、ファイルを開いたときにファイルの全データをロードし、保存時には全データをストアするというような場合には、特に上記のメソッドはオーバーライドする必要はないだろう。一方、ファイルへのストレージを併用するような場合だと、書き込みはもちろん、読み込み時にも関連するメソッドをオーバーライドして、必要な情報だけを取り込むということを実現しなければならない。また、文書ファイルが複数のファイルで構成されるような場合でも、writeToFile あたりのメソッドをオーバーライドする必要がある。なお、writeToFile などのメソッドをオーバーライドしたときには、必要に応じて元になっているクラスの同名のメソッドを呼び出しておくことは忘れないようにしよう。

では、ファイルタイプとクリエイタを保存したファイルに設定するには、writeToFile の引数が 2 つのメソッドをオーバーライドして…と思ってなんとなくうまく動きかけたのだが、上書き保存するとファイルタイプが消えてしまう。どうやら、一時的なファイルに対してファイルタイプやクリエイタを設定するのはできているけどもそれをコピーする段階で、BSD レベルのコールを使っているせいだろうか（？～想像だが）、

情報がすべてコピーされていないような気がする。そこで、MyDocument クラスに次のように、メソッドをオーバーライドした

```
public boolean writeWithBackupToFile( String fullDocumentPath,
                    String documentTypeName, int saveOperationType) {
    boolean retVal = super.writeWithBackupToFile(fullDocumentPath,
                    documentTypeName, saveOperationType);
    int fTypeInt;
    if(documentTypeName.equals("MOSAEditor Document"))
        fTypeInt = NSHFSFileTypes.hfsTypeCodeFromFileType(" ‘rtf’ ");
    else
        fTypeInt = NSHFSFileTypes.hfsTypeCodeFromFileType(" ‘TEXT’ ");
    Integer values[] = {
        new Integer(NSHFSFileTypes.hfsTypeCodeFromFileType("‘ome9’")),
        new Integer(fTypeInt)};
    String keys[] = {"NSFileHFSCreatorCode", "NSFileHFSTypeCode"};
    NSDictionary attributes = new NSDictionary(values, keys);
    NSPathUtilities.setFileAttributes(fullDocumentPath, attributes);
    return retVal;
};
```

（上記のソース中 ‘’ の部分は実際には半角のシングルクォート）

まず最初に、一連の NSDocument で定義されたデフォルトの作業をやらないといけないので、super.writeWithBackupToFile によって、元のメソッドの呼び出しを行う。その後に、保存された文書ファイルに対して、ファイルタイプとクリエイタの設定を行っているのである。文書の sh 類に応じてファイルタイプが異なる。ファイルタイプやクリエイタは、NSDictionary クラスのオブジェクトで指定するが、タイプやクリエイタは整数値でなければならない。その整数値は、NSHFSFileTypes クラスの hfsTypeCodeFromFileType メソッド（長い！）で得られる。なお、ファイルタイプとクリエイタだけを指定したいので、NSDictionary のキーにはそれに相当する文字列だけを指定しておけばいいというわけである。

(16)Preferences の組み込み

2001 年 12 月より、NSDocument クラスを中心にした文書ファイルを扱うアプリケーションを作る Cocoa の機能を解説してきた。NSDocument を継承したクラスでいくつかのメソッドをオーバーライドすることなど、基本的なアプリケーションの枠組みはすでに解説が終わっている。今後しばらくは、いろいろなアプリケーションの付加機能的な部分を説明していきたい。まずは、アプリケーションに「環境設定」（あるいは初期設定）の機能を追加することを考える。

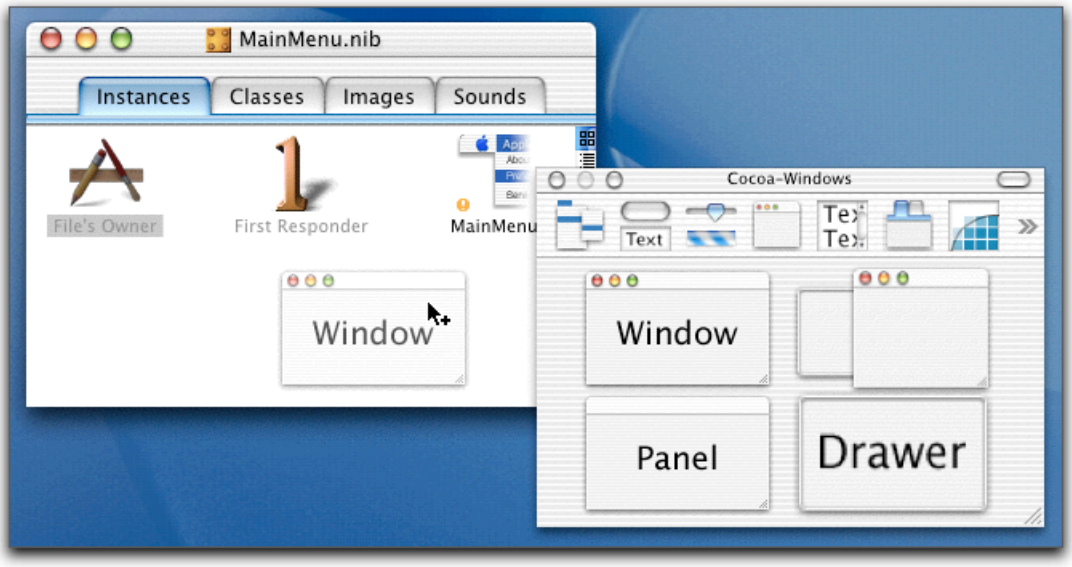
これまで説明してきた MOSAEditor というアプリケーションは、ウインドウに NSTextView を配しただけのシンプルなテキストエディタである。もちろん、いろいろなニーズはあると思うが、ここでは問題を簡単にするために、環境設定では、新規にウインドウを作成したときのウインドウの位置と大きさを環境設定で指定できるというだけにしたい。なお、NSDocumentController の機能によって、ウインドウの位置については必ずしも環境設定通りにはならないが、その点は目をつむっていただきたい。

環境設定のユーザインタフェース

MOSAEditor のプロジェクトでは、2 つの nib ファイルを使っている。1 つは MainMenu.nib であって、メニューがとりあえずは定義されている。起動時にはこの nib ファイルがロードされるので、そしてもう一つは MyDocument.nib であり、文書ファイルごとにロードされるものである。ここで、環境設定をどう管理するかはいくつかアイデアはあるだろう。たとえば、独自に nib ファイルを用意してもいいのだが、簡単な方法として、MainMenu.nib にユーザインタフェースの定義を行っておく。つまり、環境設定のインスタンスを MainMenu.nib で定義しておけば、アプリケーションに 1 つだけのインスタンスが登場することになる。必要に応じて、このウインドウを表示したり、あるいは消したりといったことをすればいいということになる。ユーザインタフェースのウインドウは NSWindows で作るのであるが、ここでは、環境設定のウインドウを NSWindows を継承して定義することにする。また、環境設定値は static なメンバを使って記録しておき、他のオブジェクトから比較的容易に参照できるという方法もやってみた。

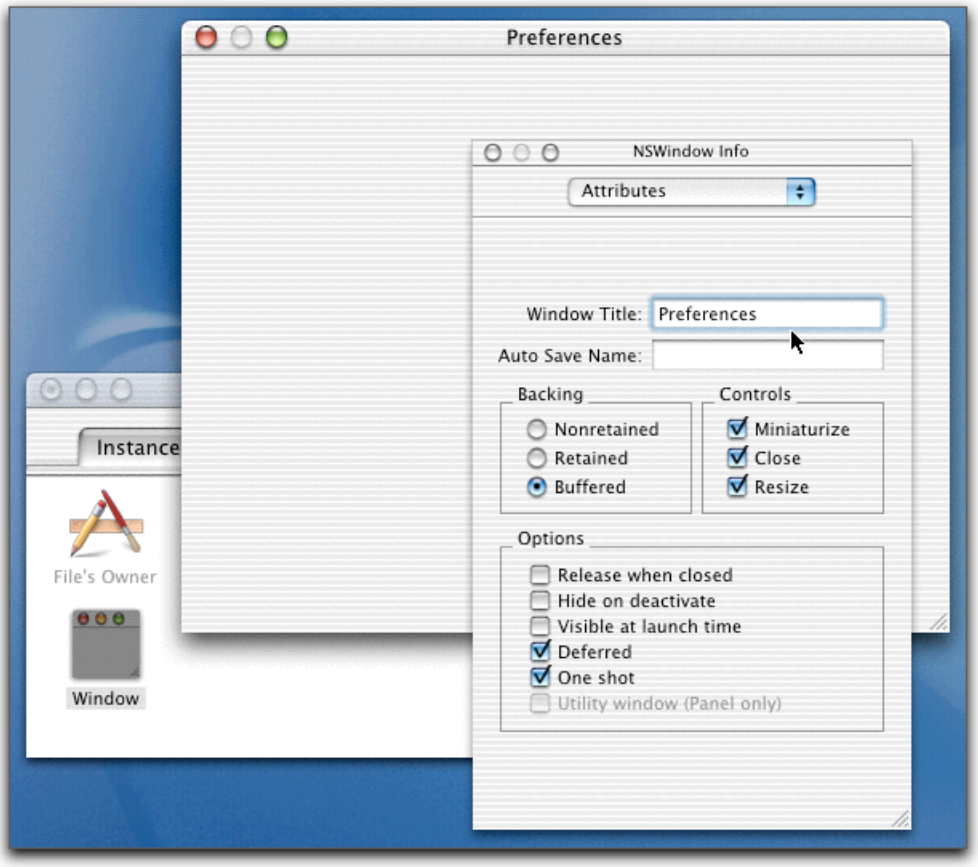
まず、MainMenu.nib を開く。そのインスタンスとして Window を追加する。ツールパレットでは Cocoa-Window を選択して、Window とかかれた大きなアイコンを、Instance のタブが選択された MainMenu.nib の文書ウインドウにドラッグ&ドロップして追加する。

MainMenu.nib にウインドウを追加する



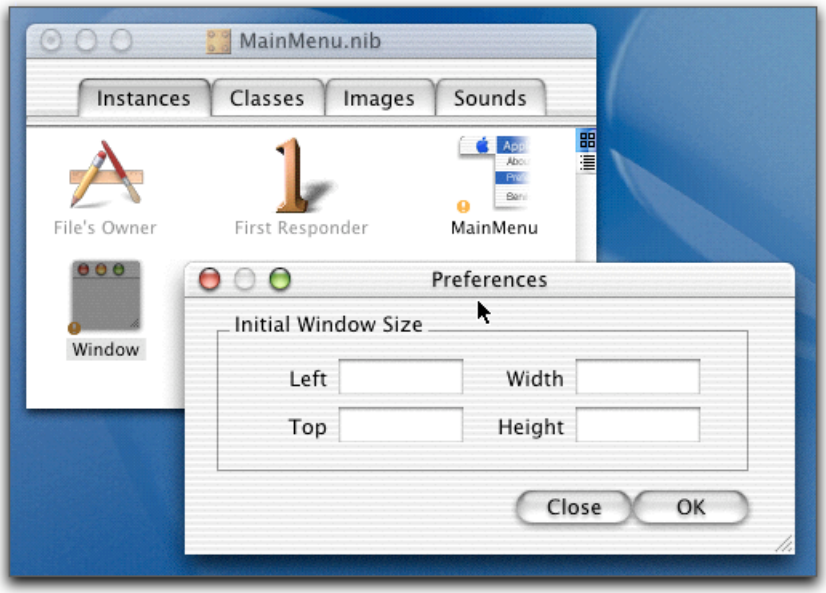
作成した Windows のアイコンが選択されている状態か、あるいはウインドウを開いおく。そして、Tools メニューの Show Info（Command+shift+I）選択して Info パレットを表示して Attributes の設定を行う。とりあえずウインドウのタイトルは Preferences にしておく。そして、起動したときにこのウインドウが開かないように Visible at launch time のチェックボックスははずしておく。

ウインドウのタイトルと起動時に開かない設定を行う



ウインドウの中には最終的には次のように 4 つの NSTextField と 2 つのボタンが存在するようなものを作った。テキストフィールドを囲むボックスなどは、いちおう見栄えの問題であるが、プログラムとリンクするのはテキストフィールドとボタンだけである。

環境設定ウインドウのユーザインタフェースを作成した



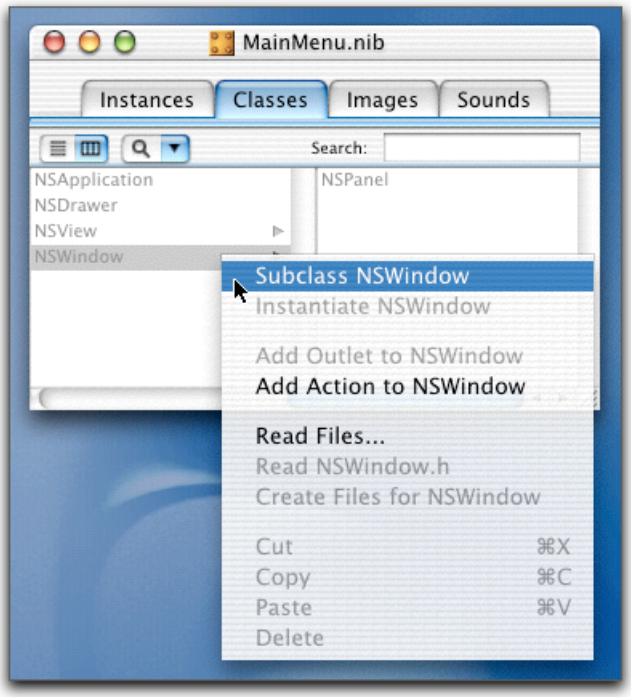
ちなみに囲っているボックスは、ツールバーの左から 6 つ目のアイコンを選択して、Cocoa-Container のパレットにして、そこにある「Box」と書かれたものをドラッグ&ドロップしてくるのが良い。ちなみに、最初から分かっていればまずは Box を持ってくるところだが、後から持ってきた場合は、Layout メニューの Send to back を選択して重なりを背後にしたほうが後の作業はしやすいだろう。

Left などの固定文字については、Attributes で右揃えにするなどしている。いずれにしても、ガイド表示があるので、きれいに並べるのはほんとにラクチンだ。

環境設定を扱うクラスの設計

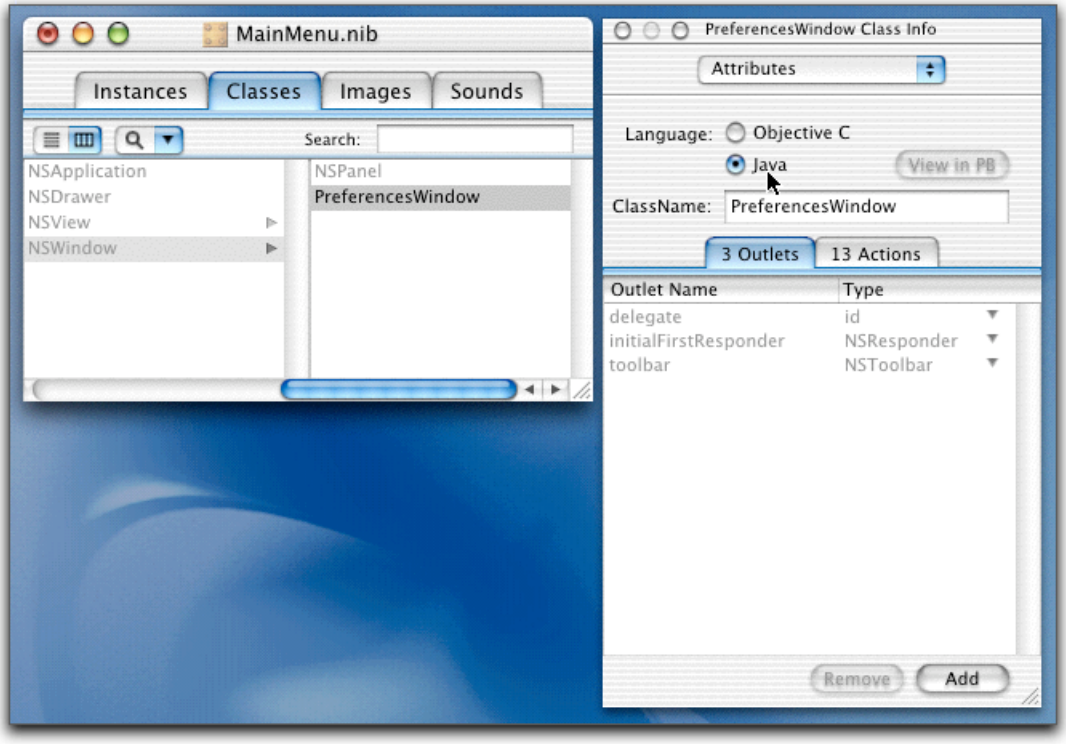
続いて環境設定を行うクラスの設計を行おう。ここでは、Cocoa に最初から用意されているウインドウを管理するクラスの NSWindows を継承する。まず、MainMenu.nib のウインドウで、Classes のタブを選択する。そして、NSObject→NSResponder→NSWindows というふうに階層をおりていき、NSWindows を control キーを押しながらクリックする。そして、表示されるメニューで Subclass NSWindow を選択する。

NSWindow のサブクラスを作成する



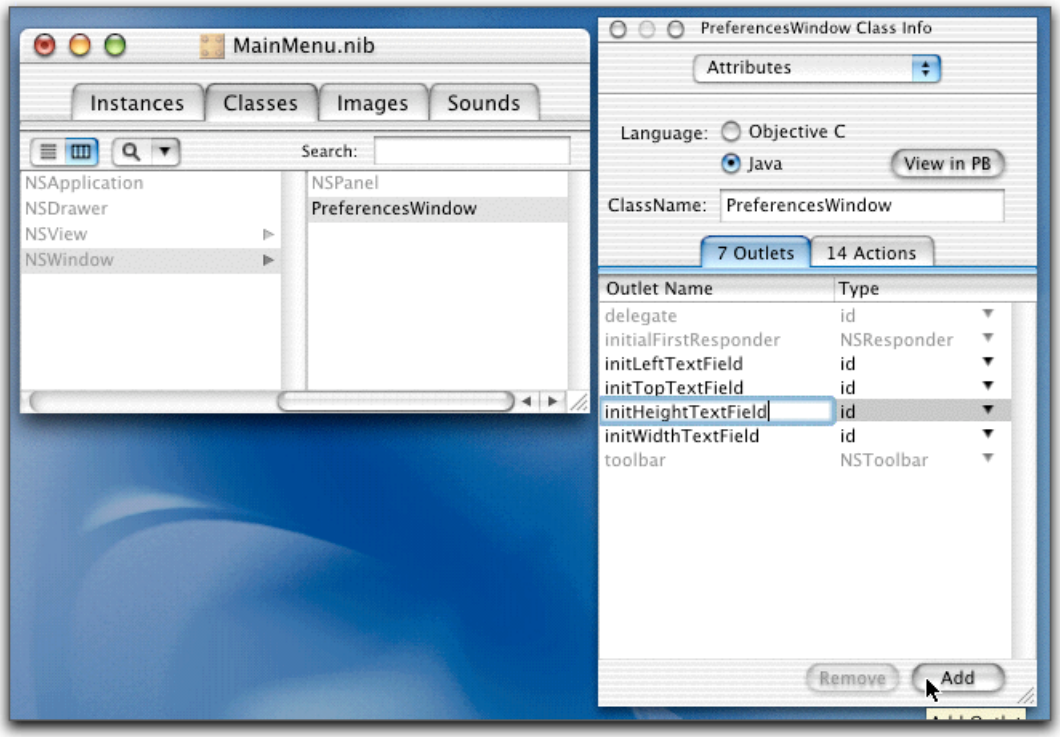
作成したサブクラスは、PreferencesWindow というクラス名にした。ここで、名前を変更するとともに、クラス名を選択して Info パレットを表示し、クラスの情報を表示する。今回の一連のプログラムは Java で作成しているので、language は Java を選択しておく。

作成したクラスのクラス名を設定し、Java を選択する



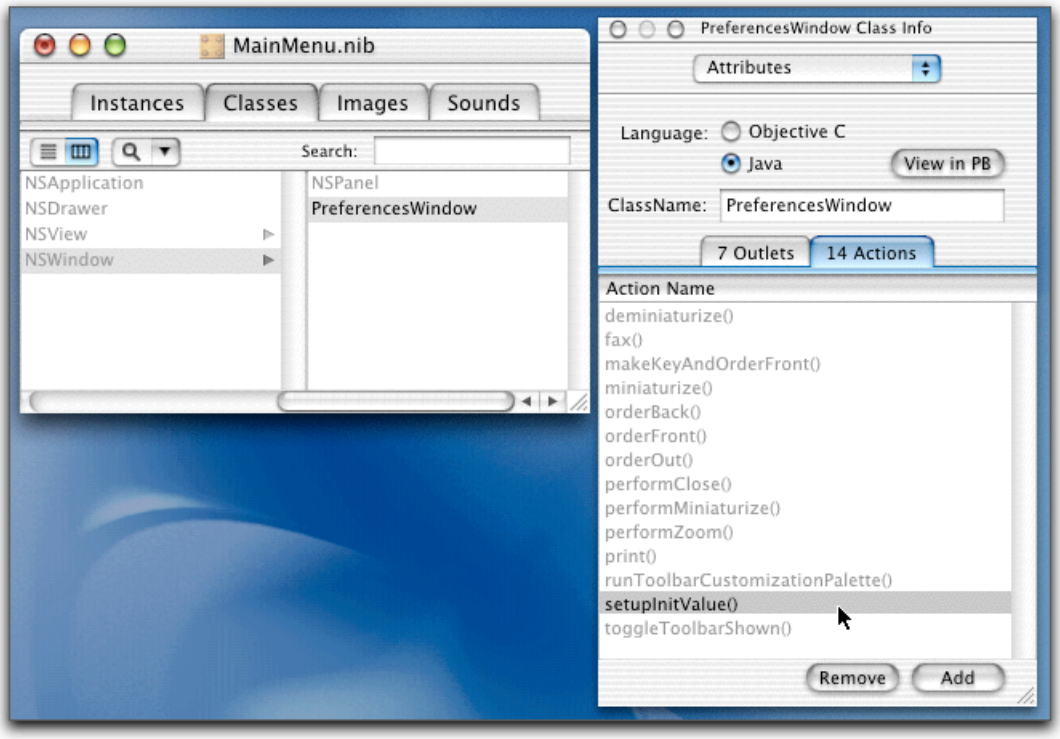
このクラスで、環境設定のユーザインタフェースを管理する。当然ながらテキストフィールドに入力された値を知りたいので、4 つのテキストフィールドそれぞれを参照するための Outlet を定義する。Outlet のタブを選択して、add ボタンをクリックすると Outlet が追加されるので、そこで、initLeftTextField などと名前を変更する。ここでは Type は id のままで良い。

テキストフィールドを参照するための Outlet を 4 つ定義する



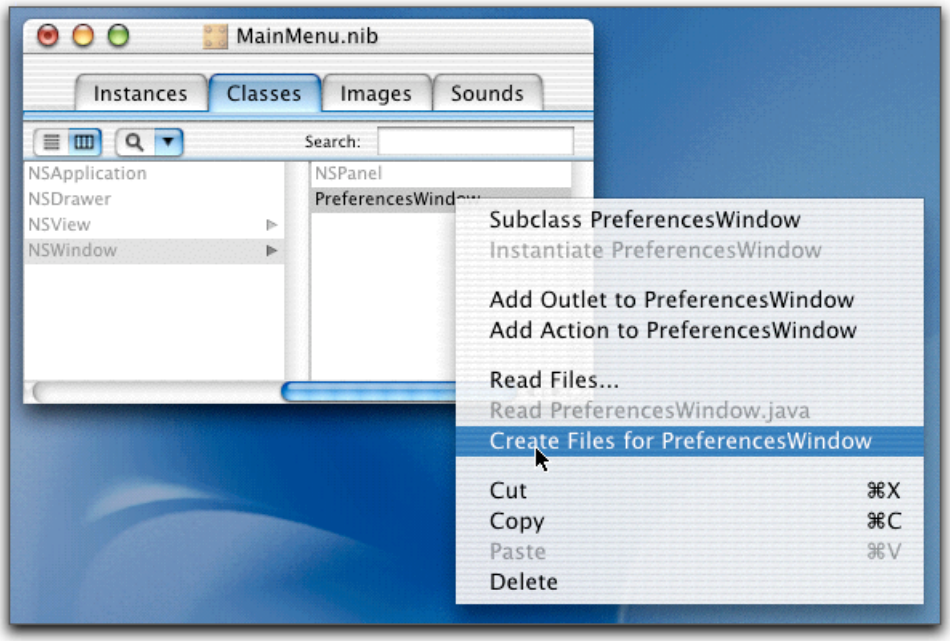
ボタンは 2 つあるのだが、Close ボタンは単にクローズするだけにするので、特にプログラムは組まない。ここでは、OK ボタンをクリックするとテキストフィールドにある値を記録するようにしたいので、OK ボタン向けに Action を 1 つ定義した。

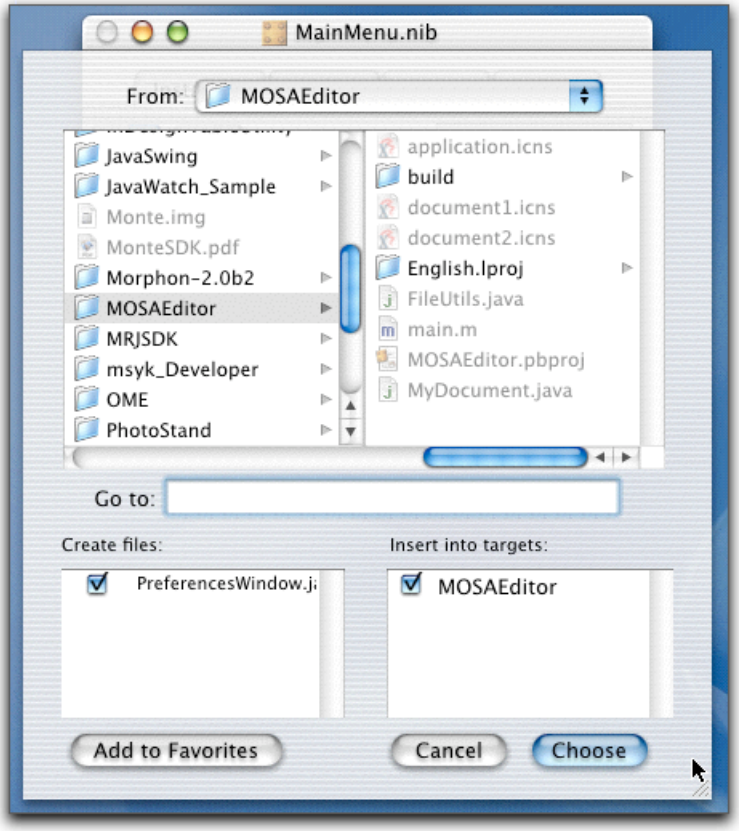
OK ボタン用の Action を 1 つ定義する



そして、この PreferenceWindow クラスのソースを作成する。クラス名を control+クリックして、表示されるメニューから Create files for PreferencesWindow を選択する。保存する場所などを指定するシートがでてくるので、プロジェクトのフォルダとなっているのを確認して Choose ボタンをクリックする。

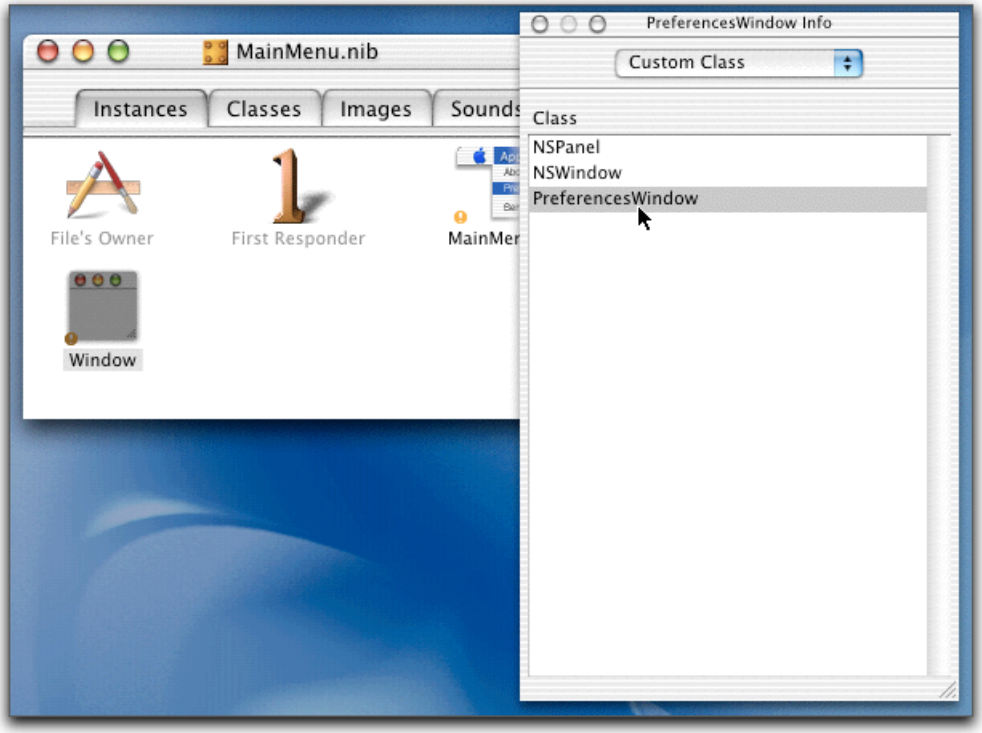
クラスのソースファイルを作成する





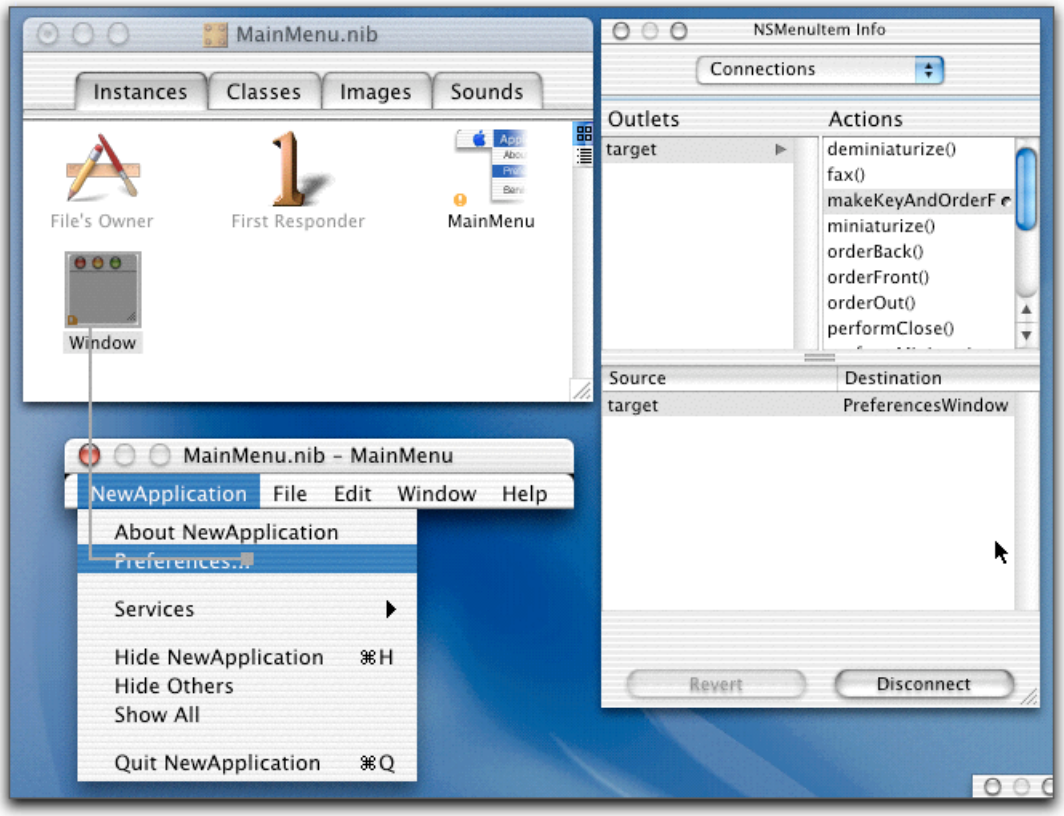
引き続いて、Interface Builder の設定を行う。次に、MainMenu.nib のウインドウで、Instances のタブを選択して、さきほど作成した環境設定のウインドウのアイコンをクリックして選択する。そして、Info パレットを表示して、ポップアップメニューで Custom Class を選択する。最初はここは NSWindow が選択されているが、PreferencesWindow を選択する。これで、環境設定ウインドウは、PreferencesWindow クラスとして生成されるようになる。

ユーザインタフェースのクラスを PreferencesWindow に変更する



アプリケーションメニューの Preferences を選択すると、環境設定のウインドウが開くように設定をしよう。ここでは、MainMenu から該当する項目を表示しておき、Preferences のところから control キーを押しながら、環境設定の Window までドラッグする。そして、Action として、makeKeyAndOrderFront()を選択して Connect ボタンをクリックする。このメソッドは、NSWindow にあるものだが、要はウインドウをいちばん手前に表示してキー入力を受け付けるというものだ。つまり、Preferences が選択されると環境設定ウインドウの makeKeyAndOrderFront メソッドが呼び出されて、環境設定ウインドウが表示されるという仕組みである。

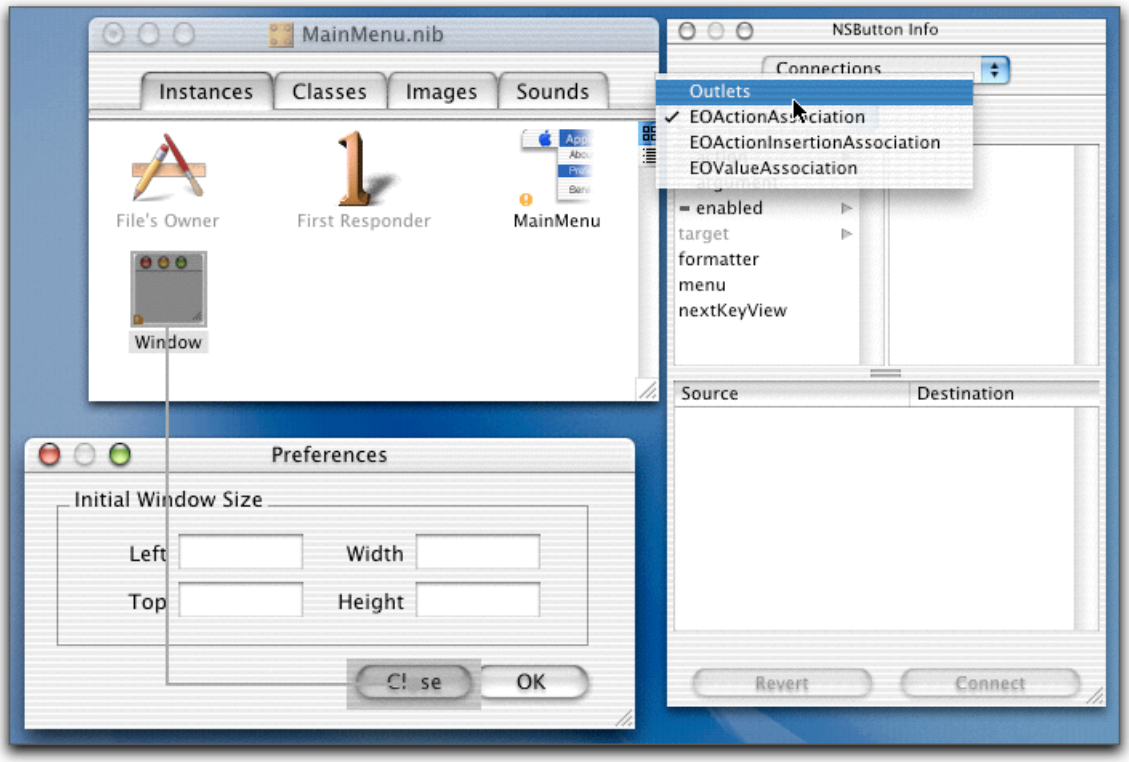
Preferences を選択すると環境設定ウインドウを表示するように設定

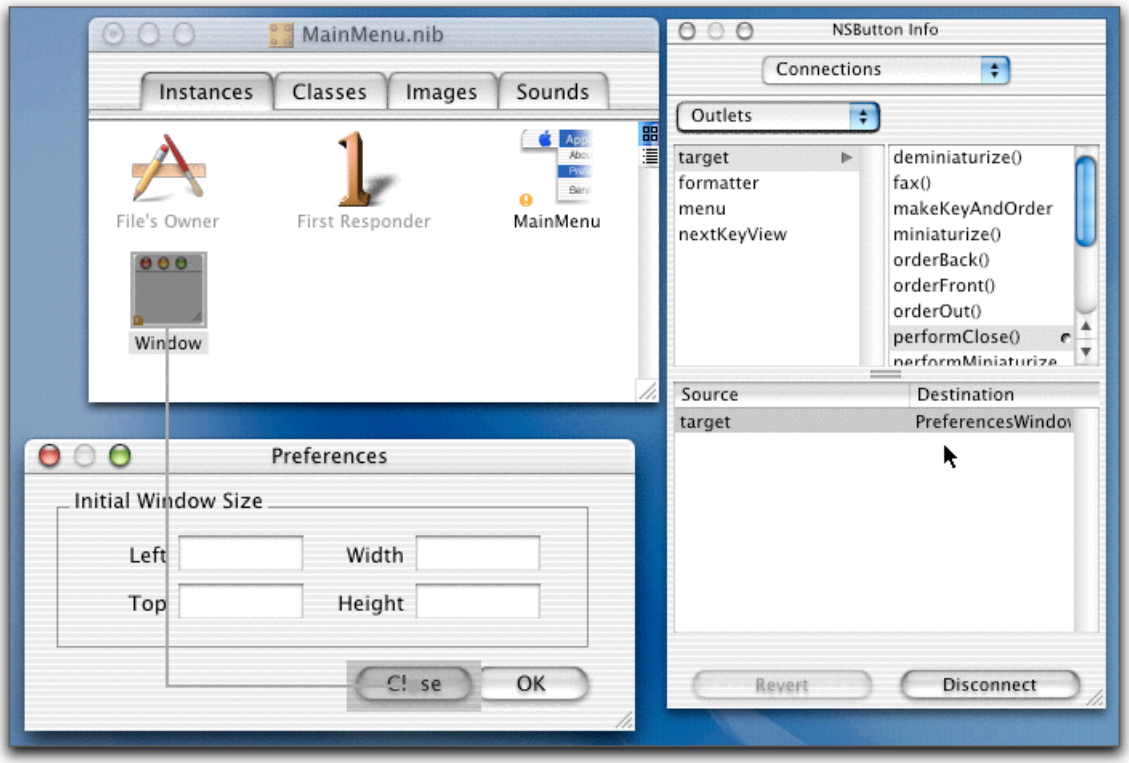


こうして、Preferences メニューから Action に接続を行うと、Preferences メニューは自動的に選択できるようになる。

次に環境設定ウインドウのコントロールからの接続を行おう。まず、Close ボタンをクリックすると、このウインドウが閉じられるようにする。そこで、Close ボタンから Window に向かって control キーを押しながらドラッグするわけだが、WebObjects がインストールされていると、Info パレットで Outlet を選択しないといけない。いずれにしても、Close ボタンの target と、ウインドウの performClose を選択する。これだけで、Close ボタンをクリックすると、ウインドウが閉じるようになる。

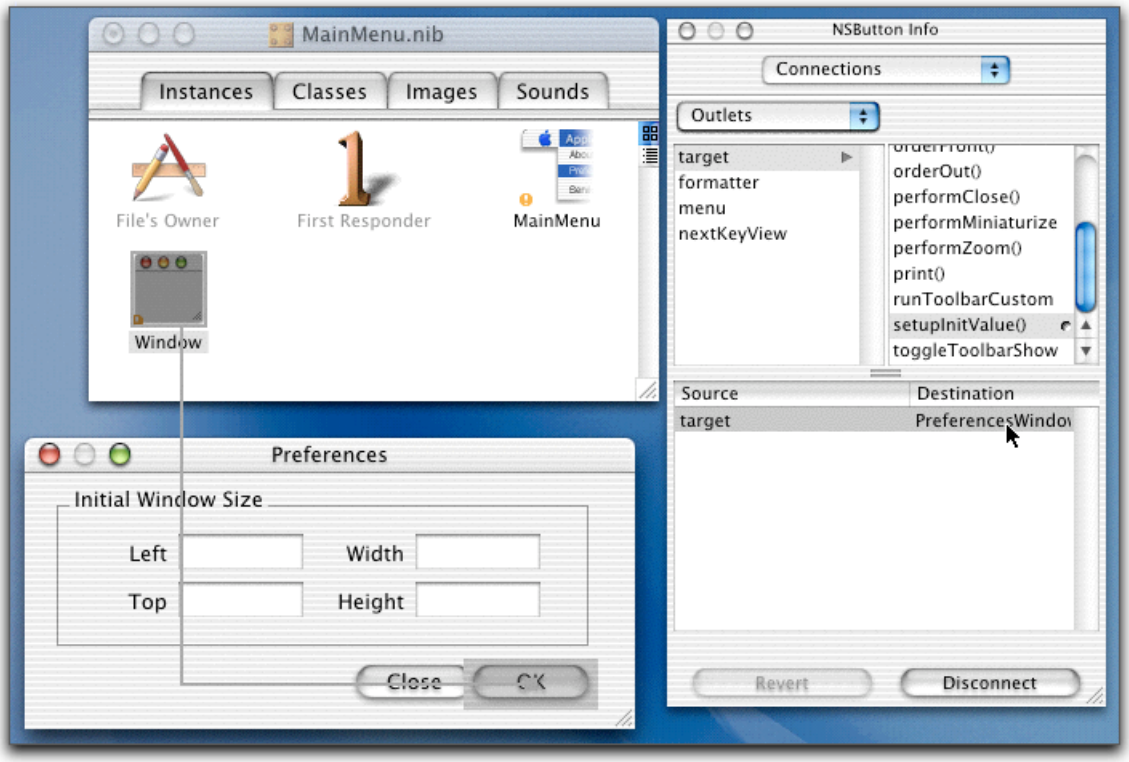
Close ボタンからウインドウの performClose メソッドを接続する





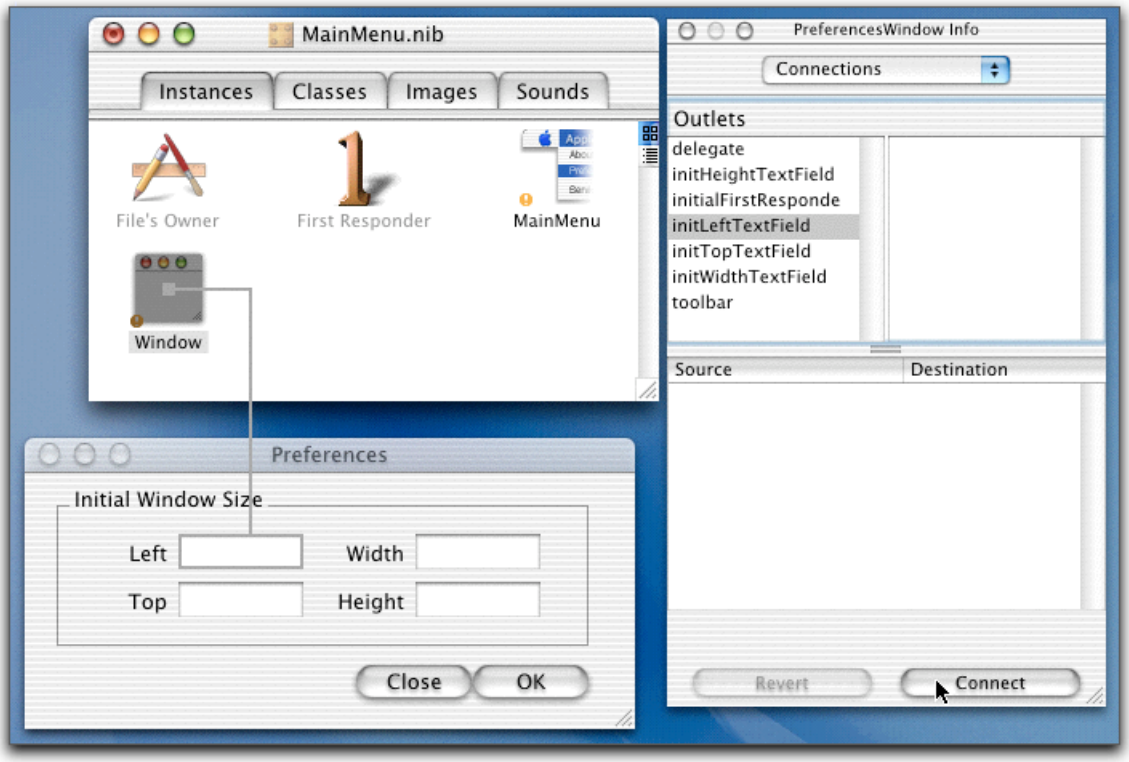
さらに、OK ボタンと、Window の間を結ぶのであるが、OK ボタンは独自に定義した Action である setupInitValue と接続しておいて、この名前のメソッドを呼ぶことにする。

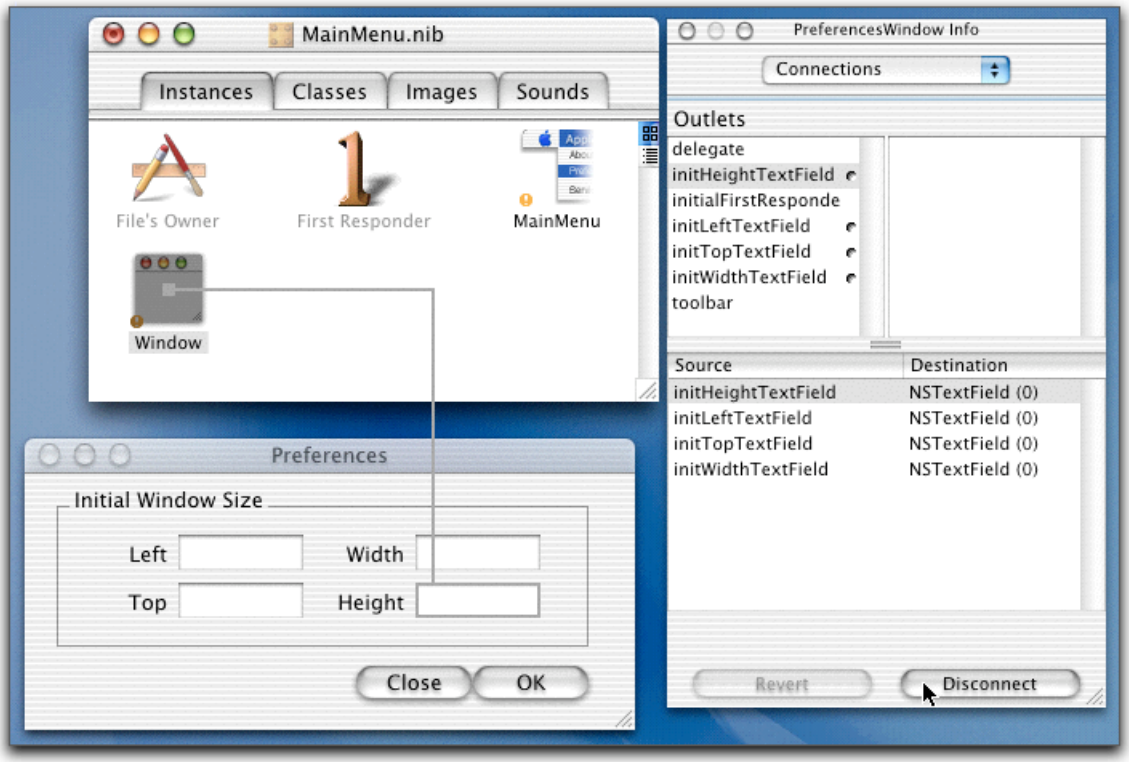
OK ボタンからは setuInitValue を接続する



さらに、PreferencesWindow のアウトレットが、ウインドウ上のテキストフィールドを参照できるようにしておく。Window から、ウインドウ上のテキストフィールドに control+ドラッグして、対応する Outlet との接続を行っておく。

PreferencesWindow の Outlet とテキストフィールドを結んでおく





とりあえず、ここまでで、Interface Builder 側の作業は一通り終わる。いちおう、流れを示したが、もちろん、実際に作るときにはいろいろと試行錯誤を行っている。いずれにしても、ポイントとなる部分が合っていれば、後はどういう手順で作業をしても

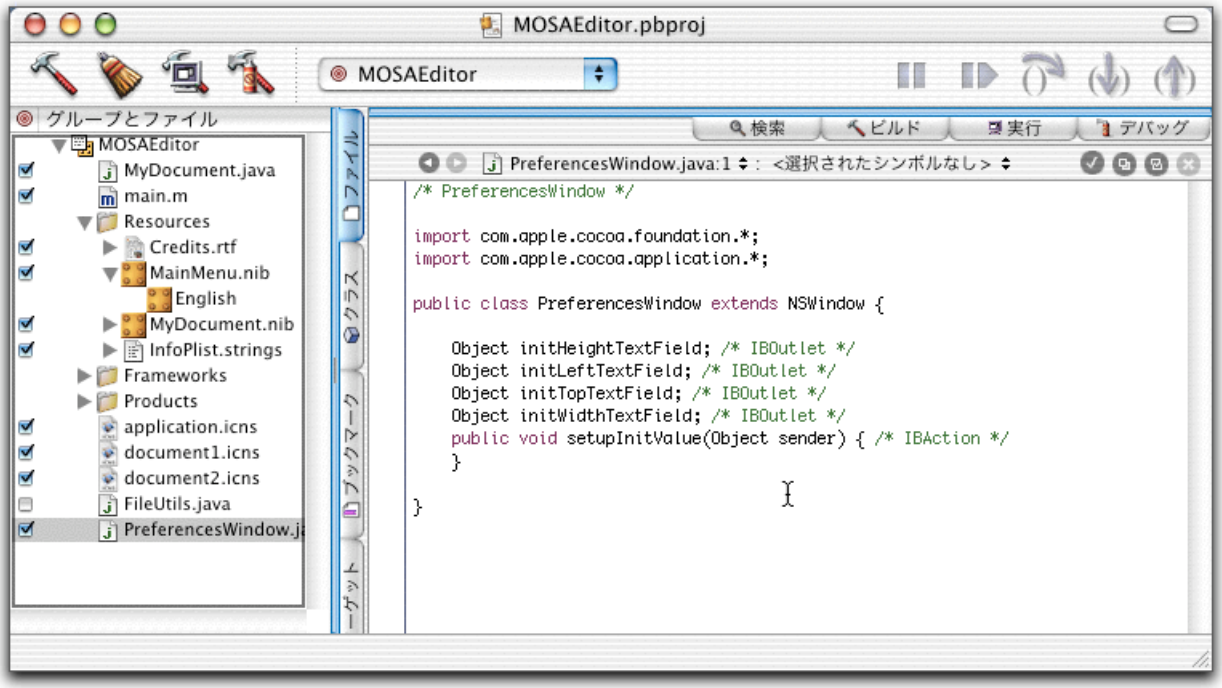
かまわないのである。

続いてクラスのプログラムを作っていくが、次回に説明を行う。

クラスを稼働させる

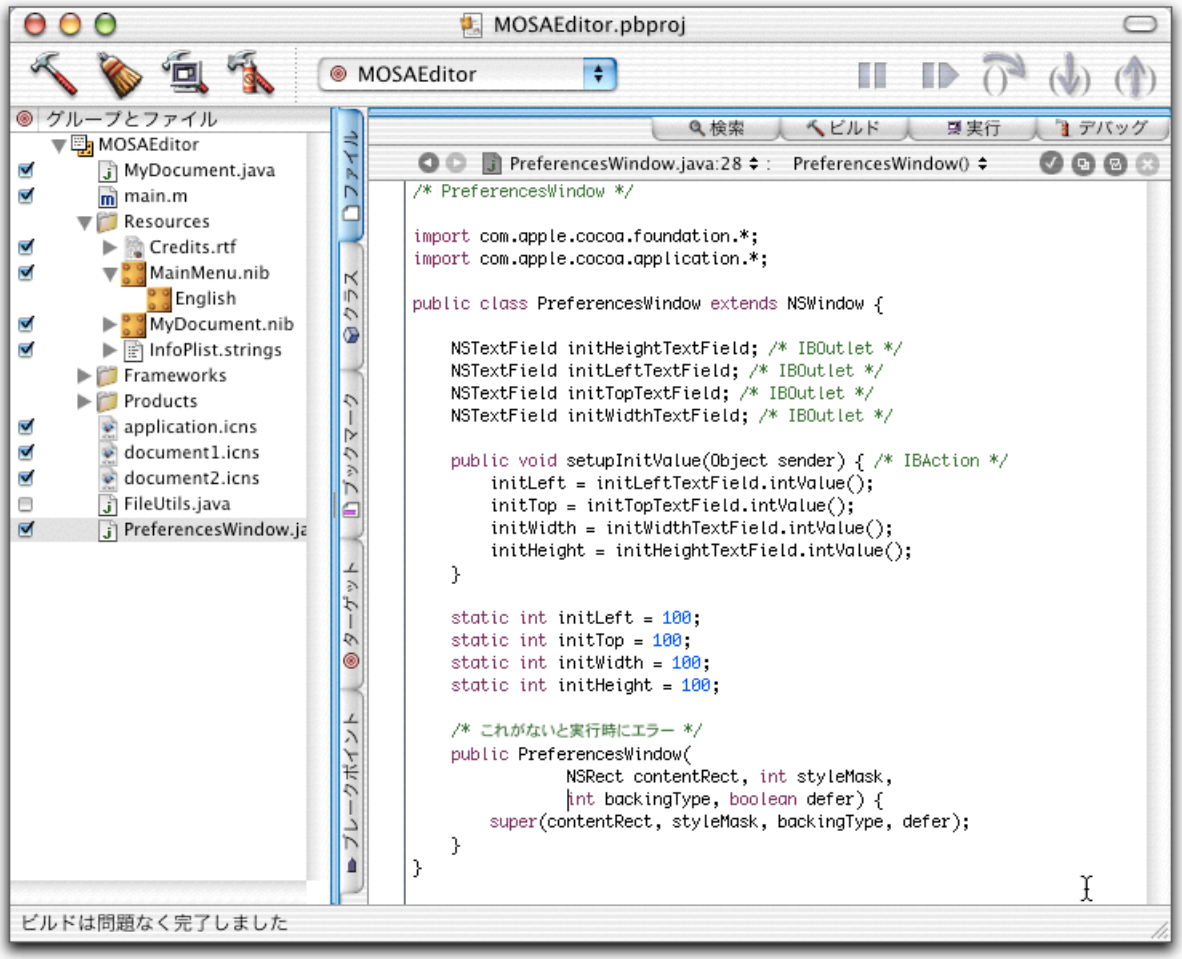
これまでのところで、MainMenu.nib に環境設定のウインドウを作成してユーザインタフェースを構築し、そのクラスのソースファイルを生成した。生成したクラスファイルの PreferencesWindow.java を Project Builder で見てみると、次の図のように、4 つの Outlet に対応したメンバー変数と、1 つの Action に対応したメソッドが出来上がっている。

生成された PreferencesWindow.java



このソースにプログラムを以下のように追加する。順次説明をしていこう。

プログラムを追加した PreferencesWindow.java



まず、Outlet に対応したメンバー変数であるが、Interface Builder にソースを生成させた段階では、変数のクラスは Object になっている。Object はどんなオブジェクトでもいわばワイルドカードのようなものなのであるが、Java ではキャストをしないといけないなどの煩わしさがあるので、ここでは、Object ではなく NSTextField クラスの変数として書き換える。こうしておいても、問題なくウインドウ上のテキストフィールドを参照できるようだ。

次に、環境設定ウインドウで設定した値を記憶するための static な変数を定義する。それぞれ、int 型で initLeft、initTop、initWidth、initHeight という変数を定義し、ここでは初期値をとりあえず適当な値にしている。なお、NSWindow などの Cocoa のコンポーネントの座標値は本来は float になっているが、そういう細かいアプリケーションなので、そのあたりは目をつむっていただきたい。なお、システムに初期設定値を記憶させる方法は、別の回に改めて説明しよう。

そして、OK ボタンをクリックして呼び出されるメソッド setupInitValue では、テキス

トフィールドから読み取った値を、static 型の変数に記憶している。たとえば、高さのテキストフィールドは、initHeightTextField で参照できるので、intValue メソッドでテキストフィールドに入力された値を整数値として得る。それを initHeight 変数に記憶させておくのである。こうすれば、他のインスタンスから、環境設定の値を参照できるというわけだ。（ところで、OK ボタンとしたが、むしろ「設定」ボタンの方がこうした動作の場合には適切かもしれない）なお、文字をテキストフィールドに入れた場合などのエラー処理は行っていない。

ここまでで、環境設定のユーザインタフェースを作り、それをウインドウとして表示して、値を保存できると言いたいところだが、実行すると、次のようなエラーが出るはずだ（前の図の、最後のコンストラクタがない状態で実行する〜メッセージの一部は割愛）。

> 2002-01-26 00:35:07.187 MOSAEditor[4110] AppKitJava: uncaught exception NSInvalidArgumentException (_BRIDGEUnmappedInitMethodImp: the java class PreferencesWindow does not implement any constructor that maps to the Objective C method initWithContentRect:styleMask:backing:defer:.)

どうやら、PreferencesWindow に必要なコンストラクタが組み込まれていないということのようだ。ここで、当然ながら、Java なので、引数のない PreferencesWindow コンストラクタは存在するものとして扱われるが、実行時にエラーがでているので、そのコンストラクタうんぬんではない。結果的には前の図に示したようなコンストラクタを定義するのだが、こうした根拠は実は Objective-C のドキュメントを参照しなければならない。Objective-C の AppKit の NSWindow のドキュメントを見ると、Creation のカテゴリで、- initWithContentRect:styleMask:backing:defer:というメソッドが紹介されている。エラーメッセージにもこのメソッドがマップされていないとでているが、ドキュメントでは Designated initializer であるとこのメソッドは紹介されている。つまり、Interface Builder で NSWindow あるいはそのサブクラスのインスタンスが定義されていれば、initWithContentRect:styleMask:backing:defer:というイニシャライザメソッドが実行されてオブジェクトの生成を行うということである。
この事実を Java に置き換えると、Interface Builder で定義された NSWindow ないしはそのサブクラスは、そこコンストラクタのうち、以下のものを呼び出すことで、イン

スタンス生成が行われるということになる。Java Bridge のドキュメントでは、Java のコンストラクタによって Objective-C でのイニシャライザが呼び出されることが明記されていることと、エラーメッセージにでているメソッドの引数並びから判断して、以下のコンストラクタが呼び出されるものと判断できるわけだ。

```
public NSWindow( NSRect contentRect, int styleMask, int backingType, boolean defer)
```

したがって、NSWindow を継承した PreferencesWindow でも、同じ引数並びのコンストラクタを定義しておく必要がある。単に定義して、同じように NSWindow 側、つまり super のコンストラクタを呼び出しておけばそれでいいわけだ。
ただ、こうしたことが、Cocoa-Java のドキュメントには明記されていないのはやはりなんとかしてほしいところである。

続いて、実際にウインドウを表示するところで、環境設定ウインドウで指定した位置とサイズにするということをしたいが、1 つの方法は、NSDocument クラスを継承している MyDocument クラスにある windowControllerDidLoadNib メソッドで、ウインドウのサイズを変更することである。たとえば、次のようなプログラムになるだろう。setFrame メソッドで、生成されているウインドウの位置やサイズを、環境設定のウインドウを管理しているクラスから取り出した数値で設定している。

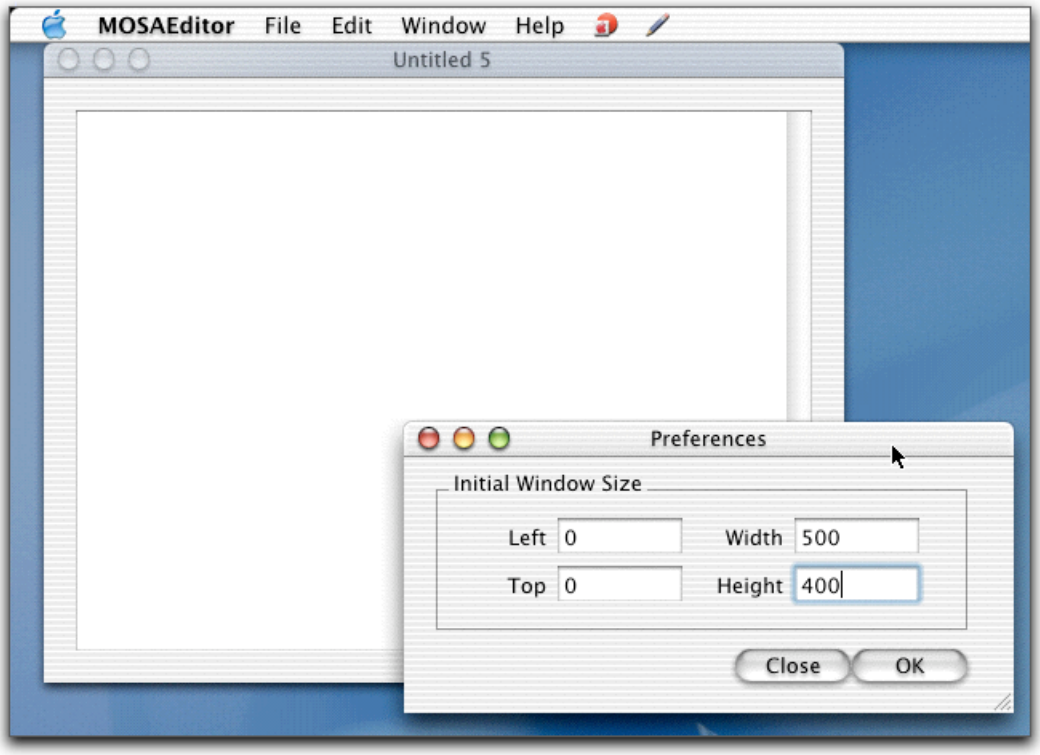
```
public void windowControllerDidLoadNib(NSWindowController    aController) {
    super.windowControllerDidLoadNib(aController);
    setupWindowFromData();

    NSRect initialRect = new NSRect(
                            PreferencesWindow.initLeft,
                            PreferencesWindow.initTop,
                            PreferencesWindow.initWidth,
                            PreferencesWindow.initHeight);
    docTextView.window().setFrame(initialRect, true);

    docTextView.window().makeFirstResponder(docTextView);
```

```
        isNibLoaded = true;
    }
```

つまり、MyDocument.nib ファイルをロードして、その定義に従ってクラスのインスタンス化が行われた後に、windowControllerDidLoadNib が呼び出される。つまり、文書ウインドウは生成されているのである。それは、ここでは NSTextView の window メソッドから参照できる。これに対して大きさを設定しているのであるが、環境設定ウインドウで入力している値は static な変数で参照できるようにしあるので、クラス名と変数名の記載で環境設定値が利用できるということである。ここで、static にしていることで簡単にアクセスできるようになるということである。

それでは実際に実行させてみよう。起動して、アプリケーションメニューから Preferences を選択する。すると、環境設定のウインドウが出るので、とりあえず Close ボタンを押してみよう。閉じるはずだが、これは NSWindows の performClose メソッドに接続しているからだ。再度開いて数値を入力し、OK ボタンをクリックする。ここでは環境設定のウインドウは閉じられるようには作っていないが、OK をクリックすることで、ウインドウの中の値を記録する。続いて、Command+N で新しいウインドウを呼び出すと、環境設定で指定した値になっているのが分かる。

MOSAEditor を動かしてみた



なお、左上位置の座標は、MOSAEditor を起動したときに表示されるウインドウにしか反映しない。これは、NSWindowController が最終的にウインドウの位置を順々に重なるように変更するからのようだ。一般的な文書作成アプリケーションではそれで問題はないだろう。もちろん、「指示通り」にしたいというニーズもあるかもしれないが、今回はそこまでの突っ込みはしないことにしよう。

環境設定の永続記憶

環境設定ウインドウで設定した初期ウインドウのサイズは、このままだと変数にセットしているだけなので、当然ながら、アプリケーションを終了すると消えてしまう。環境設定は再度起動したときに同じ値が得られていないと意味がない。一般的には、環境設定の結果をファイルに保存するのだが、Mac OS 9 まではシステムフォルダの「初期設定」フォルダにファイルを作って…ということをしていた。しかしながら、Cocoa ではそんな「ファイル処理」を一切しなくても、システムに値を永続的に記録する機能が用意されている。単に用意されているメソッドを呼び出すだけで、値を記録したり、あるいは取り出したりができるのである。おおざっぱな原理は、背後で Preferences フォルダにファイル記憶を行うのだが、そうした処理が完全に抽象化されているので

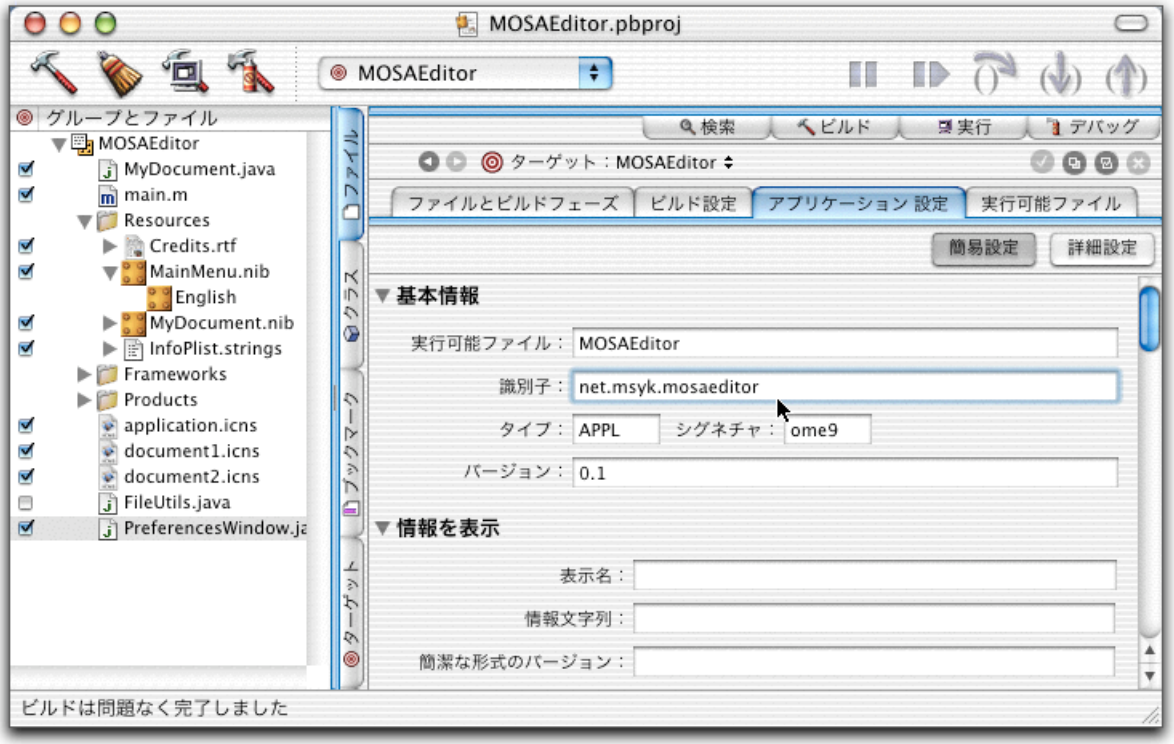
ある。これらの一連の機能は User Defaults と呼ばれており、NSUserDefaults というクラスで利用できるようになる。

◇NSUserDefaults

http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/Java/Classes/NSUserDefaults.html

まず、User Defaults を使うには、アプリケーションの識別子をきちんと設定する。識別子は基本的には何でも良いのだが、通常はドメイン名の逆順を指定するようだ。つまり、Java のパッケージのネーミングである。自分のドメインをからめた名前にしておけば、別のアプリケーションと同じ設定の競合が出るということはない。

アプリケーション設定の識別子をきちんと設定しておく



この識別子は設定の集合を識別するために使うのであるが、その背後では、設定値をファイルに保存するときのファイル名となる。Library/Preferences フォルダに、たくさんファイルがあるが、そこに「識別子.plist」というファイル名で、設定値が保存され、必要に応じて設定値が読み出される。プログラマは特にこのファイルを開いたりどうこうするということはないのだが、ファイルの存在は知っておくべきだろう。そして、実際に開くと、XML フォーマットで設定値が保存されているのが分かる。

まずは、NSUserDefaults の使い方をまとめておこう。代表的なメソッドを紹介する。クラスなのでコンストラクタを使うと思ってしまうが、NSUserDefaults は少し違っていて、すでに生成されているインスタンスを、以下のクラスメソッドでまずは取ってくる。User Defaults はアプリケーションごとというよりも、いろいろな利用範囲がある。詳細はドキュメントを見てもらいたいが、いずれにしても、以下のメソッドで得られるインスタンスは、自分のアプリケーションの識別子を持つ User Defaults が管理できる状態になっているのである。

NSUserDefaults のインスタンスを得る（Static）
NSUserDefaults NSUserDefaults.standardUserDefaults();

戻り値 NSUserDefaults のインスタンス

次に、値の実際の保存や取り出しは以下のようなメソッドを使うのであるが、まず原則は、値には「名前」をつけるということだ。名前をキーにして、取り出しを行う。従って、名前が違っていれば、複数の値を設定できるということである。なお、識別子と名前と両方あるように思うかもしれないが、識別子は（名前, 値）のセットを複数記録したものの集合体を指すものである。

値の設定や取り出しはデータの種類ごとに用意されているが、基本データは、基本データ向けのメソッドを使う。オブジェクトを記憶するときには setObjectForKey を使うが、取り出しには形式に応じていくつかのメソッドが用意されている。

User Defaults に値を名前を付けて設定する
void 〈NSUserDefaults〉.setIntegerForKey(int value, String defaultName)
void 〈NSUserDefaults〉.setLongForKey(long value, String defaultName)
void 〈NSUserDefaults〉.setBooleanForKey(boolean value, String defaultName)
void 〈NSUserDefaults〉.setDoubleForKey(double value, String defaultName)
void 〈NSUserDefaults〉.setFloatForKey(float value, String defaultName)
void 〈NSUserDefaults〉.setObjectForKey(Object value, String defaultName)

引数	value	設定する値
	defaultName	設定値につける名前

User Defaults から値を取り出す
int 〈NSUserDefaults〉.integerForKey(String defaultName)
long 〈NSUserDefaults〉.longForKey(String defaultName)
boolean 〈NSUserDefaults〉.booleanForKey(String defaultName)
double 〈NSUserDefaults〉.doubleForKey(String defaultName)
float 〈NSUserDefaults〉.floatForKey(String defaultName)
Object 〈NSUserDefaults〉.objectForKey(String defaultName)
String 〈NSUserDefaults〉.stringForKey(String defaultName)
NSData 〈NSUserDefaults〉.dataForKey(String defaultName)
NSArray 〈NSUserDefaults〉.arrayForKey(String defaultName)
NSDictionary 〈NSUserDefaults〉.dictionaryForKey(String defaultName)

戻り値	引数の名前に対応した取り出した値	
引数	defaultName	名前

User Defaults に設定してある値を削除する
void 〈NSUserDefaults〉.removeObjectForKey(String defaultName)

引数	defaultName	削除対象の名前

User Defaults に値をまとめて設定する
void 〈NSUserDefaults〉.registerDefaults(NSDictionary dictionary)

引数	dictionary	名前と値をセットにしたオブジェクト

以上のメソッドにはいずれも識別子の指定はない。アプリケーションの中で、上記のメソッドを使う限りは、識別子はアプリケーションに設定されたものが使われるのである。つまり、プログラマは、Project Builder を使う上での設定を行うだけで、後は何も気遣いは必要ないというわけである。

なお、基本データ型の取り出しは、integerForKey を使うのであるが、もし、名前が定義されていない場合には、値として 0 を戻す。これだと、値が定義されていないのか、あるいは 0 が定義されているのか、判断がつかない。そのような場合には、objectForKey メソッドを使って戻り値を調べる。もし、null が戻されたら、その名前の値は定義されていないことになる。

registerDefaults はまとめて値をセットするときには便利なメソッドだ。これらの使い

方は実際のプログラムで示そう。

環境設定値を記録する

User Defaults を使う、NSUserDefaults クラスの機能についてはすでに解説したが、結果的に、その機能を使って、PreferencesWindows.java のソースは以下のように作成した。ソースを見ながら変更点を解説しよう。

```
/* PreferencesWindow */

import com.apple.cocoa.foundation.*;
import com.apple.cocoa.application.*;

public class PreferencesWindow extends NSWindow {

    NSTextField initHeightTextField; /* IBOutlet */
    NSTextField initLeftTextField; /* IBOutlet */
    NSTextField initTopTextField; /* IBOutlet */
    NSTextField initWidthTextField; /* IBOutlet */

    public void setupInitValue(Object sender) { /* IBAction */
        initLeft = initLeftTextField.intValue();
        initTop = initTopTextField.intValue();
        initWidth = initWidthTextField.intValue();
        initHeight = initHeightTextField.intValue();

        NSUserDefaults ud = NSUserDefaults.standardUserDefaults();
        ud.setIntegerForKey(initLeft, "Position Left");
        ud.setIntegerForKey(initTop, "Position Top");
        ud.setIntegerForKey(initWidth, "Width");
        ud.setIntegerForKey(initHeight, "Height");
    }

    static int initLeft ;
```

```
    static int initTop ;
    static int initWidth ;
    static int initHeight ;

    static          {
        NSUserDefaults ud = NSUserDefaults.standardUserDefaults();
        if(ud.objectForKey("Position Left") == null)   {
            Object obj[] = {new Integer(20), new Integer(100),
                                    new Integer(400), new Integer(200)};
            Object keys[]
                = {"Position Left", "Position Top", "Width", "Height"};
            ud.registerDefaults(new NSDictionary(obj, keys));
        }
        initLeft = ud.integerForKey("Position Left");
        initTop = ud.integerForKey("Position Top");
        initWidth = ud.integerForKey("Width");
        initHeight = ud.integerForKey("Height");
    }

    /*  これがないと実行時にエラー  */
    public PreferencesWindow(
                NSRect contentRect, int styleMask,
                int backingType, boolean defer)     {
        super(contentRect, styleMask, backingType, defer);
        setDelegate(this);
    }

    public void windowDidBecomeKey(NSNotification aNotification)    {
        initLeftTextField.setIntValue(initLeft);
        initTopTextField.setIntValue(initTop);
        initWidthTextField.setIntValue(initWidth);
        initHeightTextField.setIntValue(initHeight);
    }
}
```

まず、User Defaults に記録する値の名前を、"Position Left", "Position Top", "Width", "Height"いうことにしておく。特にネーミングの規則はないようであるが、適当に自分で規則を作っておけばよいだろう。
環境設定ウインドウの OK ボタンをクリックすると、setupInitValue メソッドが呼び出される。ここでは、テキストフィールドの値をクラス変数に記録したが、ついでに、User Defaults にもそれらを書き込んでおけば、OK ボタンをクリックすることで、User Defaults に確実に記録されることになる。
一方、アプリケーションを起動したときには、記録されている User Defaults を読み出したい。その方針として、アプリケーション内ではクラス変数で環境設定の値を管理しているので、static イニシャライザで、User Defaults からの取り出しを行い、アプリケーション起動時に確実に処理されるようにした。読み出しを行えばいいのだが、integerForKey メソッドで数値で得たい。もちろん、文字列で得てから処理するとうい手もあるが、ここでは、1 つの名前に対する値が存在するかどうかを判断して、それがない場合には、はじめて User Defaults を利用するものと判断し、値とキーのセットを配列で定義して、registerDefaults メソッドを使って NSDictionary クラスで必要な値のセットをまとめて設定した。その後、User Defaults から読み出しを行っている。

このままだと、アプリケーションを起動し、環境設定ウインドウを呼び出すと、環境設定のウインドウにあるテキストフィールドは空欄のままだ。つまり、環境設定のウインドウを表示したときに、現在のクラス変数の値を各テキストフィールドに記入する必要がある。そのために、ウィンドウのデリゲートを利用する。NSWindow は、キーウインドウとなったときに、デリゲートされているオブジェクトに対して windowDidBecomeKey メソッドの呼び出しを行う。ここでは、PreferencesWindow のコンストラクタで、setDelegate(this);によって、まず、自分自身をデリゲート先に指定した。そして、アプリケーションメニューの Preferences を選択すると、環境設定ウインドウがキーウインドウになる（このウインドウの makeKeyAndOrderFront メソッドを呼び出すようになっているからだ）。したがって、PreferencesWindow クラスの windowDidBecomeKey メソッドが呼び出されるという仕組みである。ここで、クラス変数の値をテキストフィールドにセットすればよい。

繰り返しとなるが、以上のリストで、アプリケーションの識別子「net.msyk.mosaeditor」はまったくでてこない。アプリケーション設定をしておき、前回に説明したメソッドを使う範囲では、識別子は自動的に認識されて、それを元に設定がされるのである。

なお、User Defaults に設定された値は、defaults というコマンドで参照することができる。ここでは識別子は net.msyk.mosaeditor であるので、

```
% defaults read net.msyk.mosaeditor
```

とコマンドを与えれば、設定されている値が以下のように参照できる。

```
{Height = 400; "Position Left" = 0; "Position Top" = 0; Width = 500; }
```

defaults コマンドの詳細は man コマンドのマニュアルで見ていただくとして、設定を書き込んだり、あるいは削除したりということもできる。いずれにしても、とりあえずアプリケーションを作成している段階では、正しく書き込まれているかどうかをチェックすることは、Terminal からコマンドをたたけばできるということだ。（なお、default というコマンドもあるが、これはシェルスクリプトの switch ステートメントで使うキーワードだ。）

また、自分のホームフォルダにある Library/Preferences に、アプリケーションの識別子から名前をとった「net.msyk.mosaeditor.plist」というファイルがある。ダブルクリックすると、Developer Tools でインストールされる Property List Editor で、ファイルの内容が参照できる。中身は XML 形式であるが、このアプリケーションを使うと階層表示で内容を参照することもできる。

User Defaults の結果を残すファイルを参照した

環境設定ウインドウのユーザインタフェースの組み込みと、User Defaults を使った永続記憶の方法を 5 つの記事でお届けしてきたが、次回は Cocoa ならではの機能を紹介しよう。メニューを追加するだけで、いきなり高機能（！？）ワープロになってしまうというところだ。

（このシリーズの、これ以降の記事は、記事ごとに PDF を公開しています。）